Goal-oriented Hardware Design

by

Man Ping Grace Chau

B.Eng.(Hons), The Chinese University of Hong Kong (2006)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

Author ....
Department of Electrical Engineering and Computer Science
17 July, 2008

Certified by.......
Steve Ward
Professor
Thesis Supervisor

Accepted by ....
Terry P. Orlando
Chair, Department Committee on Graduate Students

Goal-oriented Hardware Design

by

Man Ping Grace Chau

Submitted to the Department of Electrical Engineering and Computer Science
on 17 July, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis presents *Fide*, a hardware design system that uses Goal-oriented programming. Goal-oriented programming is a programming framework to specify open-ended decision logic. This approach relies on two fundamental concepts—*Goals* and *Techniques*. Goals encode decision points and Techniques are scripts that describe how to satisfy Goals. In Fide, Goals represent the functional requirements (*e.g.*, addition of two 32-bit binary integers) of the target circuit. Techniques represent hardware implementation alternatives that fulfill the functions. Techniques may declare their own subgoals, allowing a hierarchical decomposition of the functions. A *Planner* selects among Techniques based on the Goals declared to generate an implementation of the target circuit automatically. Users' preferences can be added to generate circuits for different scenarios: for different hardware environments, under different circuit constraints, or different implementation criteria *etc.* A Beta processor is implemented using Fide. The quality of the implementation is comparable to those optimized manually.

Thesis Supervisor: Steve Ward
Title: Professor

Acknowledgments

I would like to thank my advisor, Steve Ward, for being such a wonderful mentor. He is always insightful, supportive and patient with me. It is my blessing to be able to work with him and I am very, very grateful for that.

Thanks also goes to my groupmate Justin Mazzola Paluska, the Planner creater who has taught me so much about research and writing. He is kind, intelligent and I respect him a lot. Thanks to Hubert Pham, for his random yet vital help. He was also my first project partner and helped me adapt when I arrived. Thanks to Chris Stawarz for his Python expertise.

A lot of thanks to the CSAIL community. Thanks to Alvin Cheung, an amazing friend who is simply a Wikipedia alive. Thanks to Alfred Ng, an outstanding TA for 6.823, for teaching me so much about hardware design. Thanks to my officemates: Jim Sukha, Asif Khan, Jason Ansel and Bill Thies, for the random entertainment and the sharing of their research expertise. Thanks to Eunsuk Kang, for teaching me Alloy. Thanks to Winnie Cheng, Sung Kim, Chen-Hsiang Yu, Angelina Lee for their academic advices and friendships.

Many thanks to the brothers and sisters in the MIT Hong Kong Student Bible Study Society and the Boston Chinese Evangelical Church. The Hong Kong community in MIT has also offered me great support. Their friendships are vital to maintain my sanity.

Thanks to my friends and family in Hong Kong, supporting me 12-hour-timezone away. Thanks to Cammy Poon for being my best friend over a decade and Kara Cheng for her thoughtful gifts. I am extremely grateful to my parents for their endless love and encouragement. They have done so much for me and there is simply nothing I can do in return to repay them.

Most importantly, I need to thank God. He teaches me much on every aspect of life and leads me through all the difficult times. His love never fails.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The hardware design process involves decisions of choosing an implementation for each required function in the circuit. The decision process is complex: (1) the implementation has to satisfy multi-dimensional constrains *e.g.* cost, performance, heat, energy, and (2) implementation strategies depend largely on the hardware environment as well as application-specific knowledge.

For example, handheld devices demand energy-efficient hardware, but can tolerate a slower implementation. In constrast, machines specially designed for encryption must run efficiently even at the cost of a more expensive implementation. The decision criteria for an implementation differs in different scenarios. In the case of implementing addition in hardware, we can choose a ripple carry adder if the cost is the most important constraint. Otherwise, a Kogge-Stone adder [1] can be used if performance is critical.

The properties of an implementation change when it is ported to a different hardware environment. This is because the building blocks (*e.g.* the logic gates) have different properties (*e.g.* area), and/or the architecture (*e.g.* the interconnection between the components) differs. The same function might require a different implementation in order to match the users' design constraints.

Designing hardware for different applications or hardware environments is currently done manually. It is desirable to be able to evaluate and compare the existing implementation recipes according to the design specification, and choose the appropriate ones to generate an implementation automatically. A framework for managing the implementation recipes is desirable.

1.1 Fide: Goal-oriented Hardware Design

This thesis presents *Fide*, a hardware design system that uses Goal-oriented programming [2, 3, 4]. Goal-oriented programming is a programming framework to specify open-ended decision logic. This approach relies on two fundamental concepts—*Goals* and *Techniques*. Goals encode decision points and Techniques are alternative ways to satisfy Goals.

In Fide, the circuit implementation decision logic is encoded using Goal-oriented programming. Goals represent the functional requirements (*e.g.*, addition of two 32-bit binary integers) of the target circuit. Techniques represent hardware implementation alternatives that satisfy the Goals. Techniques may declare their own subgoals, allowing a hierarchical decomposition of the functions. Based on the Goals asserted and decision criteria of the circuit specified, the *Planner* can generate an implementation automatically, eliminating tedious trial-and-error decision processes.

For example, by changing the decision criteria from "choose the cheapest implementation" to "choose the most efficient implementation", Fide explores the implementation alternatives and changes the circuit design to fit in the constraints. On the other hand, if the underlying hardware environment changes from one type of FPGA to another, the implementation recipes are automatically re-evaluated. Based on the performance of the recipes in the new environment, the Planner selects a new implementation for the design which is optimal for the new hardware environment.

Fide provides a powerful way to reason about design. Goals represent the implementation decisions at different abstraction levels that the Planner has to make to progressively build the system. Techniques can represent any arbitrary type of building blocks as they are simply wrappers of the implementation recipes for evaluation. Therefore, reasoning about a design is no longer limited in terms of the abstraction/layer/language of the building blocks. Instead, a system is reasoned as a whole in terms of Goals, which is an abstraction layer independent of the technologies involved in the implementation. Fide enhances hardware/software co-design because both the hardware and software components are represented by Techniques and therefore reasoned about together.

Finally, Fide is useful in training hardware designers. The designers use the GUI provided by Fide to interactively examine and manipulate the decisions made by the Planner. They gain insight on how each implementation decision affects the performance of the circuit in real time, and modify the choices to further optimize the design.

1.2 Thesis Outline

The next chapter gives a simple scenario of implementing an addition circuit to show how the system works. Chapter 3 describes the design and architecture of the system through implementing the Beta processor. Chapter 4 evaluates the Beta processor generated by Fide. Chapter 5 details the implementation. Chapter 6 presents the related work. Finally, chapter 7 outlines future work and concludes.

Chapter 2

Fide Overview

Hardware design in Fide consists of three steps: specification, evaluation and generation of the implementation. In this chapter, a circuit that adds two 32-bit binary integers is implemented to demostrate the ideas.

2.0.1 An addition circuit—a motivating example

A Goal is a parameterized decision point that describes what function is needed without specifying how to implement that function [4]. In this example, a circuit that adds two 32-bit binary integers is represented by the Goal `Addition(width=32, hardware_database=Handler, cost_vs_delay=1)`. `Addition` names the Goal, which represents the function. `width=32` is a Goal parameter that restricts the semantics of the Goal. The handler to the hardware component database for querying the hardware information is passed as Goal parameter `hardware_database`. Parameter `cost_vs_delay` is passed to specify how the user weights the properties of the implementation recipes while choosing which one to use. In this example, the user set `cost_vs_delay` to 1, which means the user wants the cheapest implementation.

Goals do not specify how to implement the functions but Techniques do. Techniques that satisfy the `Addition` Goal are ripple carry adder, carry-select adder and a Xilinx-specific verilog addition implementation that only works when targeting an FPGA. The Planner resolves the `Addition` Goal by searching the above Techniques.

In order to evaluate the properties of the Techniques and choose the best one to satisfy the `Addition` Goal, the dependencies of the Techniques must be resolved. The dependencies are specified as subgoals. For example, the 32-bit ripple carry adder has 32 `FullAdder` subgoals. A Technique fails if the dependencies cannot be satisfied *e.g.* there are not enough full adders. At the same time, the Techniques access the hardware database through the Goal parameter `hardware_database` to query whether the current environment supports the corresponding implementation. A Technique fails if the hardware environment does not match the implementation.

After resolving all the dependencies, the Plan Tree is built. Evaluation starts at the leaf Techniques, where the Techniques check the properties of the basic building blocks by querying the hardware database. At each Goal point, the Planner calculates the satisfaction value for each Technique according to the satisfaction formula specified in the Goal. The Technique with the highest satisfaction is chosen to satisfy the Goal. The properties *e.g.* cost, delay of the subgoals are passed up the Plan Tree for the parent Techniques to compute their own properties. The evaluation is done when the top Goal is reached, and an implementation can be generated.

Figure 2-1 shows the Plan Tree in this example. The Verilog Technique fails since the target environment is not an FPGA as indicated in the hardware environment. Once a Technique fails, the Planner stops exploring the subgoals for the Technique. Notice the carry select adder recursively depends on subgoal of its own kind. The recursion here means the implementation of the function (*e.g.* addition of two 32-bit binary integers) depends on the same function but with different parameters (addition of two 16-bit binary integers being performed twice in parallel). The ripple carry adder is chosen because it is the cheapest implementation.

It is simple to create another implementation for a different decision criteria. For example, instead of generating the cheapest implementation, the user now wants the fastest implmenetation. This is done by changing the Goal parameter `cost_vs_delay` from 1 to 0. It means the weighting of the cost in the satisfaction formula has changed from 100% to 0. Figure 2-2 shows the resulting Plan Tree.

Fide benefits from the open-ended nature of Goal-oriented programming. Suppose

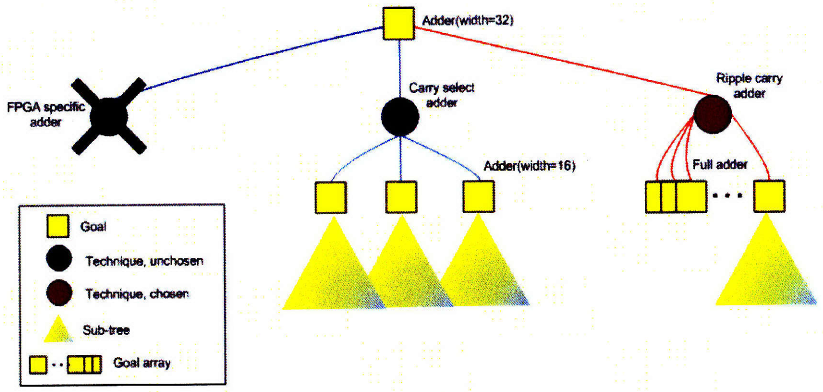


Figure 2-1: The Plan Tree that represents the decision logic for implementing an Addition circuit.

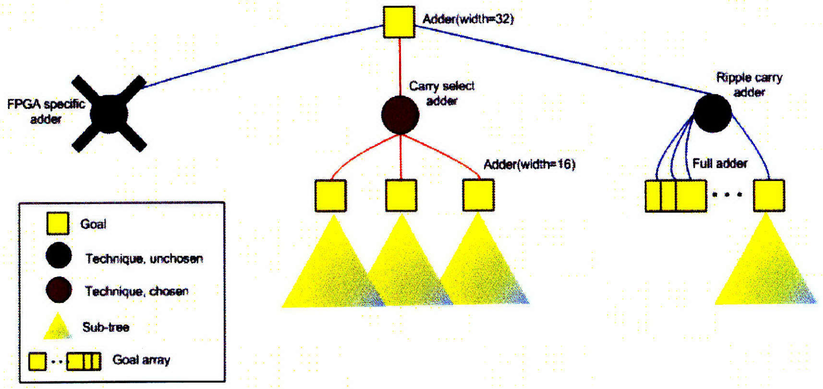


Figure 2-2: After changing the weighting from "choose the cheapest one" to "choose the fastest one", the carry select adder is chosen.

the user learns a new way to implement the addition circuit and wants to include that into the planning process. The user wraps the new implementation—the Kogge-Stone adder—as a Technique by specifying how to evaluate and instantiate the implementation. After the new Technique is added, the Planner adopts the new implementation as it is faster than the one chosen—the carry select adder. The resulting Plan Tree is shown in figure 2-3.

Goal-oriented programming enables system evolution. The functions required in the system represented by Goals are the guidelines to select the necessary building blocks. The Planner deploys better implementation alternatives for the asserted Goals as they appear to improve the system over time, without any manual redevelopment.

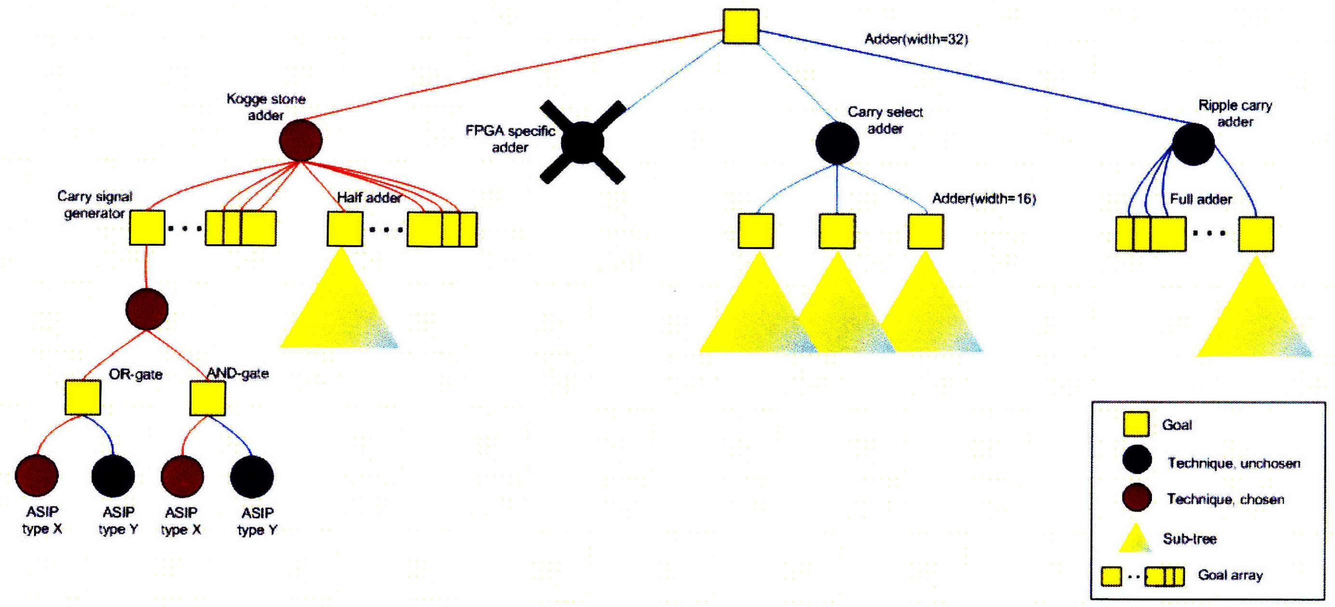


Figure 2-3: The Kogge-Stone adder is added and chosen.

2.1 Goal-oriented programming

In Fide, the hardware implementation decision logic is represented by the Plan Tree made up of Goals and Techniques.

2.1.1 Goals

Figure 2-4 shows the `Addition` Goal specification. Besides the Goal name and the Goal parameters, the Goal specification includes the Goal properties and the satisfaction formula. The Techniques that satisfy this `Addition` Goal must report Goal properties, which describe the qualities of the implementations, *e.g.* cost and delay. The satisfaction formula declares what properties of the Goal should be included for calculating the satisfaction, and the weighting for each properties. The user can change how the Planner calculates the satisfaction by changing those weighting parameters. The Goal parameters are divided into two types: functional parameters and non-functional parameters. The functional parameters restrict the semantics of the Goal *e.g.* `width` is a functional parameter as it further specifies the function of the Addition circuit. On the other hand, `cost_vs_delay` is a non-functional parameter as it is simply a weighting factor in the satisfaction formula.

```
name:Add
properties:delay cost
functional attributes:width=0  #default value
non-functional attributes: cost_vs_delay=0 hardware_database=None #default value
evaluation:
satisfaction=1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Figure 2-4: The addition Goal specification.

2.1.2 Techniques

Figure 2-5 shows a Technique script that implements a ripple carry adder. The first line declares the Goal that the Technique satisfies. The `via` statement names the Technique. The rest of the Technique is divided into phases. The `eval` phases are

```
to Add(width, hardware_database, cost_vs_delay):
    via ripple_carry:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        fas = subgoal_array(goal=FullAdder, number=goal.width,
                            hardware_database=goal.hardware_database,
    cost_vs_delay=goal.cost_vs_delay)

    eval:
        for fa in subgoals.fas:
            solution.delay += fa.s_delay
            solution.cost += fa.cost

    commit:
        sub_comp = {''fulladder'': subgoals.fas[0].component}
        adder = adderRCA(''rca'', component=sub_comp,
                        parameters={''width'':goal.width})
        adder.initialize()
        solution.component = adder.instantiate()
```

Figure 2-5: The ripple carry adder Technique.

imperative code that computes the properties of the recipe *i.e.* speed and area in this example. The `subgoals` phases declare the dependencies of the implementation. Finally, the `commit` phase instantiates and connects the chosen implementation to the rest of the system.

2.1.3 Plan Tree

The Plan Tree is a data structure that represents the decision logic. The Planner builds the Plan Tree by finding Techniques that might satisfy the top level Goal, which involves recursively matching the subgoals of each Technique it finds. The Plan Tree takes the form of an `AND/OR` tree since each Goal can be satisfied by any one Technique, but each Technique needs all of its subgoals satisfied. The Plan Tree enumerates all known strategies for the implementation, where a path from the root Goal node to leaf Techniques represents a particular implementation strategy [4].

The Planner is application-generic—it does not require any specific knowledge about the implementation choices to execute the decision logic. It runs the eval phases of the Techniques to expose the properties of each choice in the Plan Tree. It then chooses the best Technique for each Goal to generate the implementation. Figure 2-6 shows the Planner GUI with the chosen implementation for the addition circuit.

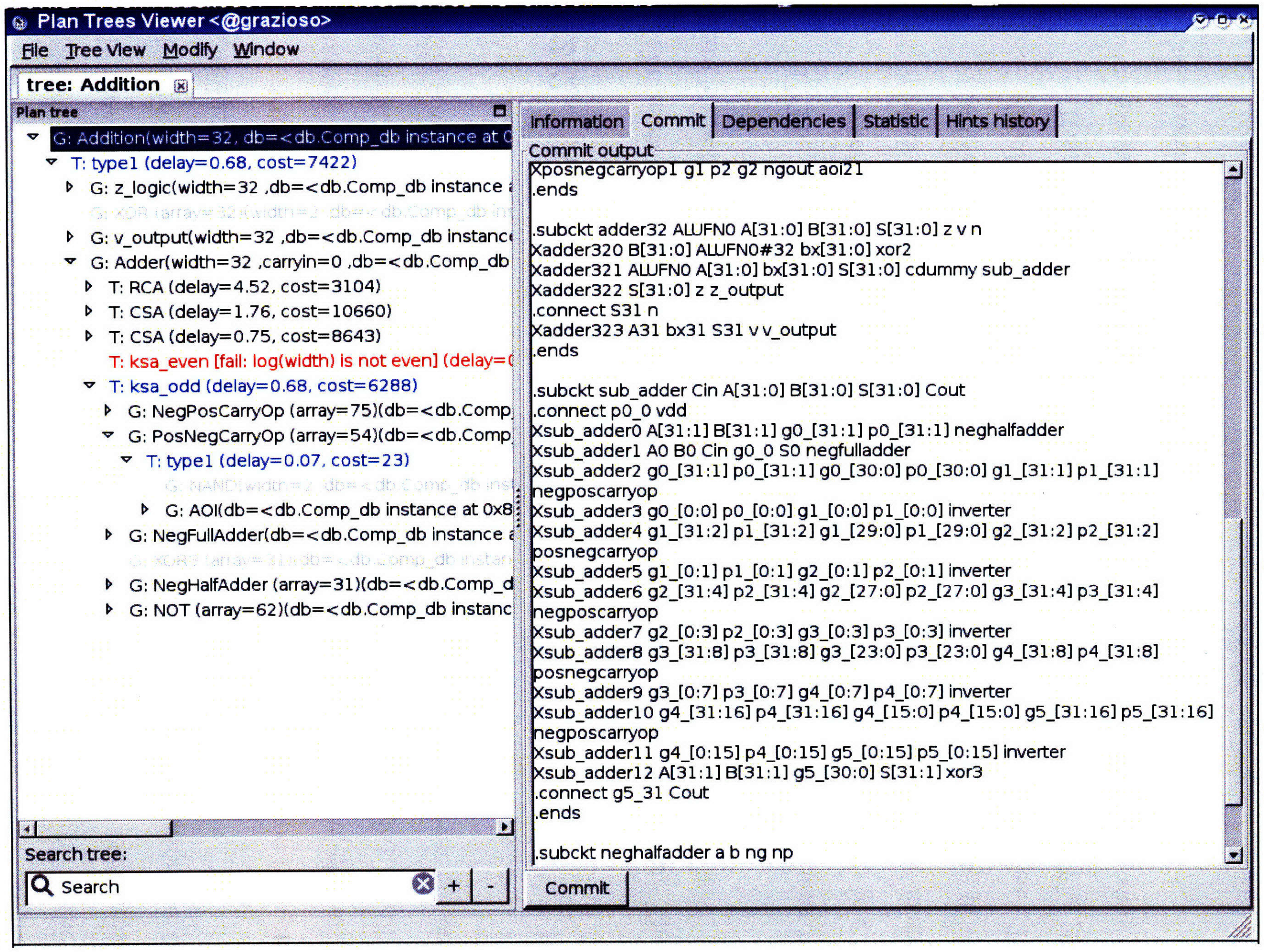
A new implementation recipe is added to the system by wrapping it in new Technique, which is evaluated by the Planner automatically. Nothing else in the existing system needs to be changed—the decision logic is truly open-ended. Old implementations, when worse than the new ones, simply "fade out" and never get chosen again.

2.2 Modifying the circuit

The users can modify the design by changing the Goal parameters. The followings are some of the examples on how the users can change the system:

1. The user can change the circuit requirements. Examples are changing the hardware environment or changing the parameters of the circuit (*e.g.* the input width). For example, instead of generating a 32-bit ALU, the users now want a 64-bit one. These are done by changing the Goal parameters of the top level Goal. The changed parameters are propagated down the tree. The affected Techniques are re-evalated again to compute new Properties. The Planner then re-selects the appropriate Techniques to generate a new design that fits the new requirements.

2. The user can change the decision criteria. For example, when porting the circuit from a general-purpose machine to a handheld device, the users want the most energy-efficient implementation instead of the fastest one. This is done by changing the weighting factors, which are the Goal parameters, in the Sat-



```
▽ G: Addition(width=32, db=<db.Comp_db instance at (
  ▽ T: type1 (delay=0.68, cost=7422)
    ▷ G: z_logic(width=32 ,db=<db.Comp_db instance a
      G: XOR (array=32)(width=2 ,db=<db.Comp_db in
    ▷ G: v_output(width=32 ,db=<db.Comp_db instance
    ▽ G: Adder(width=32 ,carryin=0 ,db=<db.Comp_db
      ▷ T: RCA (delay=4.52, cost=3104)
      ▷ T: CSA (delay=1.76, cost=10660)
      ▷ T: CSA (delay=0.75, cost=8643)
        T: ksa_even [fail: log(width) is not even] (delay=(
      ▽ T: ksa_odd (delay=0.68, cost=6288)
        ▷ G: NegPosCarryOp (array=75)(db=<db.Comp
        ▽ G: PosNegCarryOp (array=54)(db=<db.Comp
          ▽ T: type1 (delay=0.07, cost=23)
              G: NAND(width=2 ,db=<db.Comp_db ins
            ▷ G: AOI(db=<db.Comp_db instance at 0x8
        ▷ G: NegFullAdder(db=<db.Comp_db instance a
          G: XOR3 (array=31)(db=<db.Comp_db instan
        ▷ G: NegHalfAdder (array=31)(db=<db.Comp_d
        ▷ G: NOT (array=62)(db=<db.Comp_db instanc
```

```
Xposnegcarryop1 g1 p2 g2 ngout aoi21
.ends

.subckt adder32 ALUFN0 A[31:0] B[31:0] S[31:0] z v n
Xadder320 B[31:0] ALUFN0#32 bx[31:0] xor2
Xadder321 ALUFN0 A[31:0] bx[31:0] S[31:0] cdummy sub_adder
Xadder322 S[31:0] z z_output
.connect S31 n
Xadder323 A31 bx31 S31 v v_output
.ends

.subckt sub_adder Cin A[31:0] B[31:0] S[31:0] Cout
.connect p0_0 vdd
Xsub_adder0 A[31:1] B[31:1] g0_[31:1] p0_[31:1] neghalfadder
Xsub_adder1 A0 B0 Cin g0_0 S0 negfulladder
Xsub_adder2 g0_[31:1] p0_[31:1] g0_[30:0] p0_[30:0] g1_[31:1] p1_[31:1]
negposcarryop
Xsub_adder3 g0_[0:0] p0_[0:0] g1_[0:0] p1_[0:0] inverter
Xsub_adder4 g1_[31:2] p1_[31:2] g1_[29:0] p1_[29:0] g2_[31:2] p2_[31:2]
posnegcarryop
Xsub_adder5 g1_[0:1] p1_[0:1] g2_[0:1] p2_[0:1] inverter
Xsub_adder6 g2_[31:4] p2_[31:4] g2_[27:0] p2_[27:0] g3_[31:4] p3_[31:4]
negposcarryop
Xsub_adder7 g2_[0:3] p2_[0:3] g3_[0:3] p3_[0:3] inverter
Xsub_adder8 g3_[31:8] p3_[31:8] g3_[23:0] p3_[23:0] g4_[31:8] p4_[31:8]
posnegcarryop
Xsub_adder9 g3_[0:7] p3_[0:7] g4_[0:7] p4_[0:7] inverter
Xsub_adder10 g4_[31:16] p4_[31:16] g4_[15:0] p4_[15:0] g5_[31:16] p5_[31:16]
negposcarryop
Xsub_adder11 g4_[0:15] p4_[0:15] g5_[0:15] p5_[0:15] inverter
Xsub_adder12 A[31:1] B[31:1] g5_[30:0] S[31:1] xor3
.connect g5_31 Cout
.ends

.subckt neghalfadder a b ng np
```

Figure 2-6: The Planner GUI. The left panel shows the Plan Tree, with the chosen Techniques in blue and failing Techniques in red. The Goal parameters and Properties of the Techniques are also shown. The right panel shows the implementation generated.

isfaction formula of the Goals. After re-evaluation, the Planner selects the Techniques that meet the current criteria most to generate the implementation.

3. The user can change the design constraints. For example, because of the budget limit, there is now a maximum cost imposed. Figure 2-7 shows a slightly more complicated Satisfaction formula. The Satisfaction formula is a Python script that computes the Satifaction value at the end. The execution environment of the Satisfaction script is made up of the Goal parameters and the Goal Properties. It states if the cost is under `min_cost`, the cost will not contribute to the Satisfaction computation. However, if the cost is greater than `max_cost`, the Satisfaction value will be set to -1. Because conventionally the Satisfaction value for any runnable Technique should be positive, setting the Satisfaction value to -1 is equivalent to failing the corresponding Technique. By changing the `min_cost` and `max_cost`, which are the Goal parameters, the users can change the constraint imposed on all the implementations of a function. After the modification, the Plan Tree starts re-evaluation as described previously.

```
name: processor
properties: MIPS cost
functional attributes: width=0
non-functional attributes: min_cost=0 max_cost=0 hardware_database=None
cost_vs_delay=0
evaluation:
if cost < min_cost:
    satisfaction = 1/delay * 100
elif cost < max_cost:
    satisfaction = 1/(delay * cost) * 100
else:
    satisfaction = -1
```

Figure 2-7: A sightly more complicated Goal specification.

2.3 Generating the implementation

If the user is satisfied with the design, she commits the Plan to generate the implementation. Committing the plan means executing the commit phases of the chosen Techniques.

In this example, committing the `FullAdder` Goal creates instances of full adders. The instantiated components are passed up to the Plan Tree. The parent Technique *i.e.* the ripple carry adder Technique connects the full adders from the subgoals to generate the addition circuit. Figure 2-8 shows how an implementation is generated from the Plan Tree. Commit starts at the leaves of the tree to satisfy the implementation dependencies of the components higher in the hierarchy.

In this thesis, the hardware is implemented in JSim[5]. JSim is chosen for its ease of use and test cases of Beta are readily avaliable as developed in 6.004[6]. To ease Technique programming, we use a Python-based hardware component library that "compiles to" JSim. The library aims to provide a well-defined interface to instantiate and connect each component, and to generate the corresponding JSim implementation after commit. As a result, programming the commit phase of a Technique is trivial as the underlying complexity of creating the hardware description is delegated to the component library. Figure 2-9 shows the JSim implementation that the Planner generates for the adder.

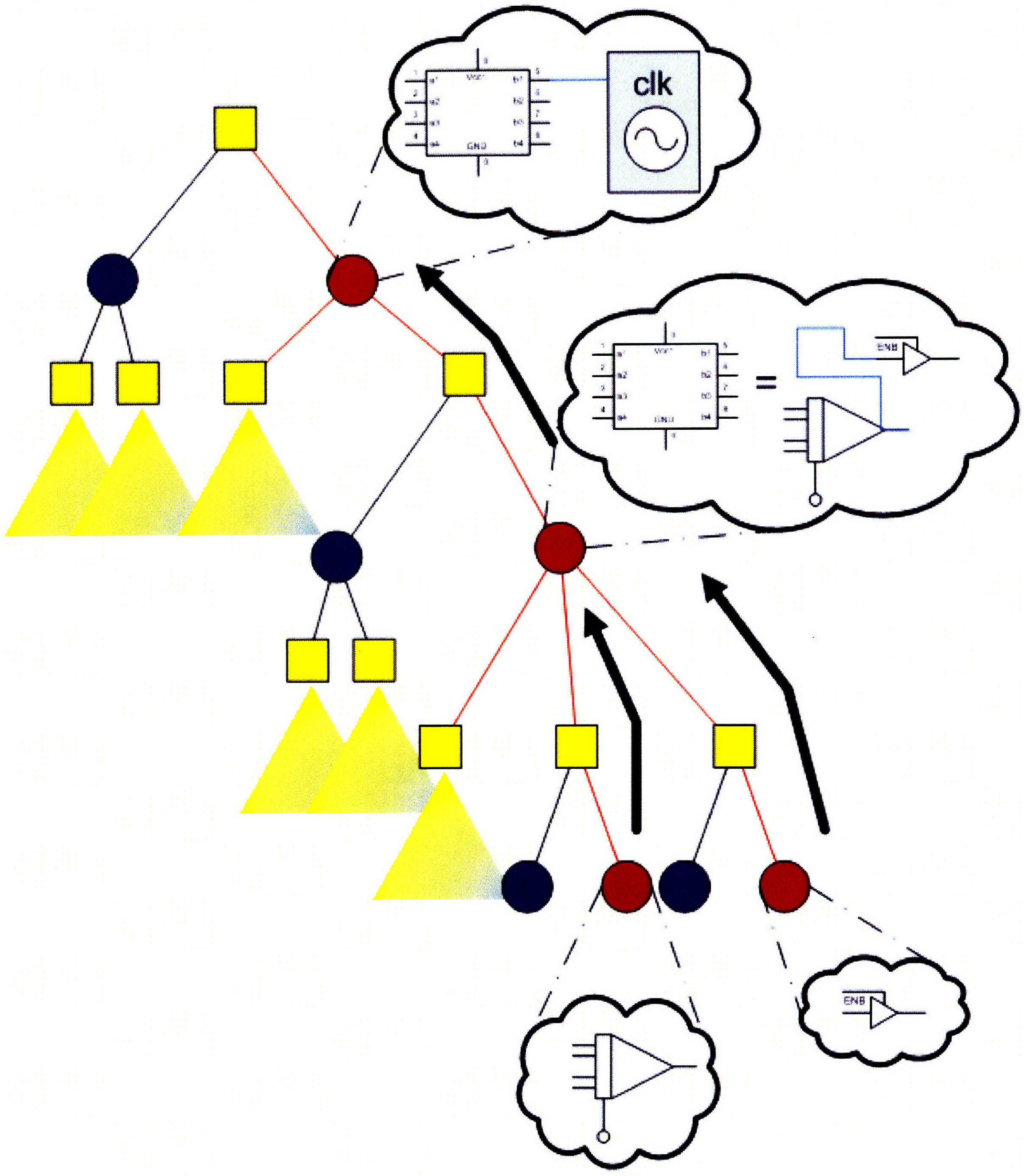


Figure 2-8: During commit, the components are instantiated and propagated up the Plan Tree.

```
.subckt fulladder a b c0 c1 s
Xfulladder0 a b g1 xor2
Xfulladder1 g1 c0 s xor2
Xfulladder2 a b g2 nand2
Xfulladder3 a c0 g3 nand2
Xfulladder4 b c0 g4 nand2
Xfulladder5 g2 g3 g4 c1 nand3
.ends

.subckt z_output S[31:0] z
Xz_output0 S[31:0] z0[7:0] nor4
Xz_output1 z0[7:0] z1[1:0] nand4
Xz_output2 z1[1:0] z nor2
.ends

.subckt adder32 ALUFN0 A[31:0] B[31:0] S[31:0] z v n
Xadder320 B[31:0] ALUFN0#32 bx[31:0] xor2
Xadder321 ALUFN0 A[31:0] bx[31:0] S[31:0] cdummy rca
Xadder322 S[31:0] z z_output
.connect S31 n
Xadder323 A31 bx31 S31 v v_output
.ends

.subckt rca C0 A[31:0] B[31:0] S[31:0] C32
Xrca0 A0 B0 C0 c1 S0 fulladder
Xrca1 A[31:1] B[31:1] c[31:1] c[32:2] S[31:1] fulladder
.ends

.subckt v_output A31 B31 S31 v
Xv_output0 A31 na inverter
Xv_output1 B31 nb inverter
Xv_output2 S31 ns inverter
Xv_output3 A31 B31 ns fir nand3
Xv_output4 na nb S31 sed nand3
Xv_output5 fir sed v nand2
.ends
```

Figure 2-9: The generated Addition circuit.

Chapter 3

Design and System Architecture

3.1 Design principle

Fide provides a framework to add, enumerate, evaluate and compare the implementation recipes systematically. Goal-oriented programming makes explicit the seperation of circuit specification and implementation. By exploiting the underspecified nature of Goals, the Planner automatically chooses among the recipes to generate the required implementation.

The users should be able to interact with Fide in three ways. The first is by asserting the Goals—the user specifies what functions are needed in a design. The second is by programming Techniques—the users specify how to evaluate and generate the circuit in Techniques. The third is by controlling how the Planner should make the decision. However, the ways that the users interact with the system might conflict with the open-ended nature of Goal-oriented programming. For example, in order to allow users to specify preferences over the choices, it is tempting to create syntax in Techniques to rank the subgoal choices. However, adding a new Technique for the subgoals would then require modification of the existing Technique scripts. The system should strive to be easy for users to program and control the decision logic, at the same time not restricting the implementation choices.

3.2 Design challanges

The followings are a few key design challanges of Fide:

Ease of Technique programming Technique programming involves two parts: specifying the how to evaluate the implementation recipe and the instantiation of the hardware components when chosen. To enhance encoding of the implementation decision logic, the new syntax are added to (1) reduce the amount of redundant codes for evaluating similar implementations, and (2) make it natural to declare hardware implementation dependencies.

The commit phase of Technique is not the place to program the circuit implementation. It is simply a placeholder to instantiate and connect the building blocks when the corresponding implementation is chosen. A JSim Component library is therefore created to bridge between the Technique and the actual implementation. This keeps programming Technique simple, without exposing the internal structure of the circuit in the Technique.

Scalability Compared to the examples in JustPlay[4], where the number of nodes in the Plan Tree is less than a hundred, Fide's Plan Trees are huge. The Plan Tree contains up to a thousand nodes when Technique evaluation goes down to the gate level. Thus, there are performance and memory problems for building, evaluating and browsing the Plan Tree.

The scalability problem involves both computation and representation issues. Brute-force search is the simplest way to find the optimal implementation. However, it is not feasible if the search space is huge. In Fide, even the performance of heuristic search is not always acceptable with such a huge Plan Tree. On the other hand, full representation of the Plan Tree takes a long time to build. The system might even run out of memory when it tries to store the data structure. Direct display of the Plan Tree in the Planner GUI also makes browsing difficult.

As such, scalability comes with the tradeoff of the quality of the implementation generated, as well as the completeness of the Plan Tree representation. It is important to strive a balance between the two.

Incremental circuit refinement The main goal of Fide is to achieve automatic circuit generation for different applications. The users should therefore be able to incrementally modify the design in order to tweak the circuit for different scenarios.

The followings are a few requirements to ease design refinement:

1. It should be easy for users to understand why the Planner made such decisions and provide sufficient information for the users to decide how to tweak the circuit.
2. It should be straightforward for the users to specify the changes *e.g.* decision criteria, parameters of the circuit (*e.g.* width of the input, the input function), the hardware environment, the constraints of the circuit, *etc.*
3. Re-evaluation of the Plan Tree should be done efficiently.

3.3 Overall system architecture

Figure 3-1 shows the overall architecture of Fide, which consists of the followings. The JustPlay paper [7] gives more details on the Planner architecture.

- The Planner which manipulates the Plan Trees. It consists of:
  - A Goal cache, to check for Goal reuse in the Tree.
  - A scheduler, to queue the tree nodes for evaluation. From the JustPlay[7] paper, it shows that the `PrecedentQueueScheduler` is the most efficient scheduler. It always queues a tree node before its parents to avoid scheduling a node redundantly.
  - A Technique database, to search for Techniques for the Goals.

  - A Technique interpreter, to translate the Technique into a Technique class. The Technique class has a standard API for the Planner to invoke evaluation and commitment of the Technique.
  - A Goal parser, to parse the Goal specification and compute the Satisfaction value for the Techniques.

- The Planner GUI which sits between the users and the Planner.

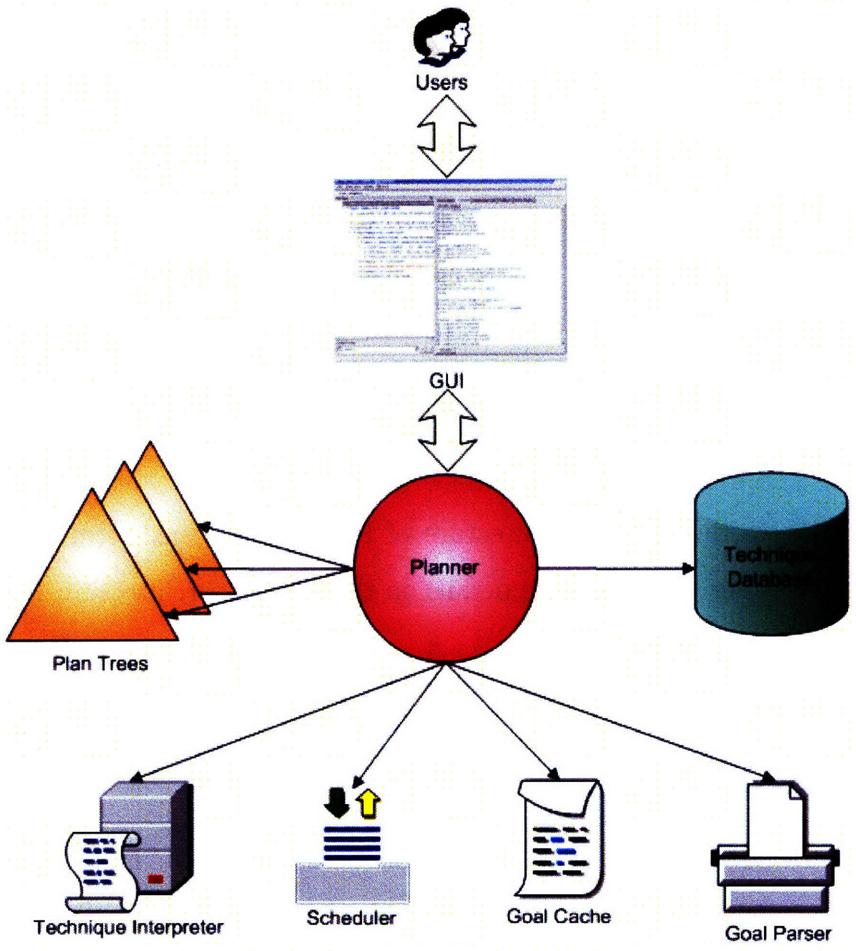


Figure 3-1: Fide architecture.

3.4 Construction and Evaluation of the Plan Tree

In this section, a Beta processor is implemented as an example to illustrate the construction and evaluation process of the Plan Tree. To be able to understand the example, the Beta architecture is presented first.

3.4.1 The Beta architecture

Figure 3-2 shows the Beta architecture and how that can be broken down hierarchically into a Plan Tree representation. The two Techniques implement the Beta in this example are the pipelined and unpipelined architecture. The pipelined architecture is a 2-stage pipeline, divided into the instruction fetch stage and everything else in the second stage. The following is the Beta subsystems:

- *The program counter.* It consists of D flip flops to store the address, the PC+4 circuit to compute the following address and a 5-way multiplexer to select the next address based on the `reset` signal and the current opcode. There are multiple ways to implement the PC+4 circuit *e.g.* a normal adder tied to 4, or an incrementer.

- *Memory.* The memory is for storing both the data and the instructions. There are multiple ways to implement the memory—for example, with different numbers of ports. The easiest implementation is to have three ports: one for instruction read, one for memory read and one for memory write. To decrease the memory size, the memory can be implmeneted by eliminating one or more ports. For example, the data read and data write can share the same port. To implement that, additional logics are needed to multiplex the data access to the memory. Eliminating the ports reduces the size of the memory as an additional port requires additional drivers and storage cells. However, it may introduce very slight delay due to the additional multiplexing logics.

- *Register file.* It is a 32-bit register with 31 slots.

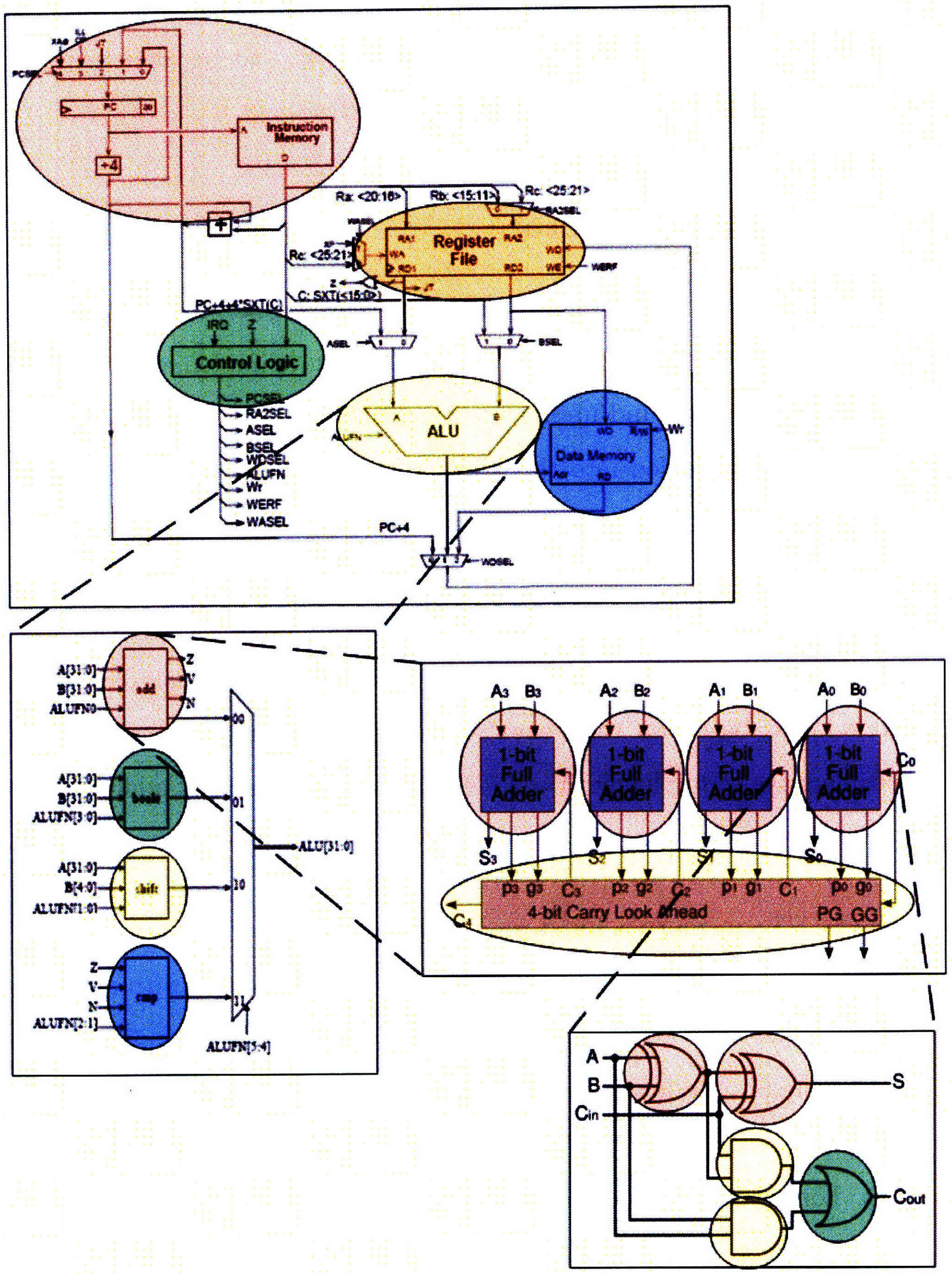


Figure 3-2: The Beta architecture. Each circle represents a subgoal. Circle of the same color represents subgoal of the same kind.

- *ALU.* The ALU consists of an adder/substractor, a shift unit, a comparator and a Boolean unit. There are at least two ways to implement the shift unit: 1. use two seperate shifters for shifting right and left, or 2. use just one shifter and add additional logic to control shifting left or right. The latter approach adds slightly more delay but is cheaper to implement.

  There are numerous ways to implement the adder/substractor. The slowest but the cheapest implementation is the ripple carry adder. The carry select adder is a faster implemention but takes up more space. The Kogge-Stone adder is a parallel prefix form carry look-ahead adder. It is fast—it can generate carry signals in O(log n) time, and is smaller than the carry select adder.

- *Control logic unit.* It generates the control signals *e.g.* write enable, PC select *etc.* based on the opcode. The easiest implementation is to generate all signals using a ROM. However, this is expensive and half of the storage is wasted because there are 64 entries (address width=6) in total in the ROM but there are only 32 valid Beta instructions. A cheaper way to implement the control logic unit is by generating some of the signals using logic gates.

- Other miscellaneous logics required: multiplexers for the write-back/selecting the ALU input/selecting the register address, logics for computing the branching address, logics for controlling the supervisor bits *etc.*

3.4.2 Construction and evaluation

Figure 3-3 shows the Plan Tree construction and evaluation flow chart. Initially, the Planner discovers that the `Beta` Technique implements the `processor` Goal from the Technique database. In order to compute the Properties of the `Beta` Technique, the Planner has to resolve the implementation dependencies for `Beta`, *i.e.* finding Techniques to satisfy its subgoals. The subgoals of `Beta` are the ALU, the data memory, the register file, the program counter and some logic gates that generate the control signals. While the Planner is searching the Techniques to recursively satisfy the subgoals, it discovers some building blocks can be reused. For example, there are multiple adders having the same input bits in a recursive carry-select adder architecture. The Planner realizes the redundant components from its "hardware environment cache". More details on the cache will be discussed in section 3.4.5. As a result, the Planner would only instantiate one instance of the adder and reuse it in other part of the circuit.

The construction process ends when the Planner finishes resolving the Technique dependencies. Evaluation starts at the leaves of the Plan Tree, where it chooses the most suitable Technique to satsify each Goal. The Properties of the subgoals are copied to the `Solution` objects and are propagated up the Tree. The parent Technique, which depends on the subgoals, copies the Properties of the subgoals from the `Solution` objects to compute its own Properties. For example, the Beta pipelined architecture implementation depends on the register file, the control logic unit, the logics that compute the branching address and other components. The branching address computation can run in parallel with the control signal generation and the register data retrieval. In other words, these components form two independent datapaths. As the subgoals report their Properties, the `Beta_pipelined` Technique realizes the two datapaths have different delay: the delay of the branching adress computation is less than the summation delay of the control logic unit and the register file. It can then determine where the critical path is and compute the overall delay for itself, which equals the summation delay of the control logic unit, the register file, the ALU and the write back logic.

When the evaluation has reached the top Goal, a Plan is chosen and the evaluation is done.

3.4.3 Technique programming

A Technique provides evaluation information for the recipe it represents to give the Planner a generic way to search for feasible implementation. It mixes both imperative and declarative style code. Programmer declares Goals to recursively decompose the implementation decision logic and writes codes to analyze the implementation choices.

To support encoding of the hardware implementation decision logic, the following

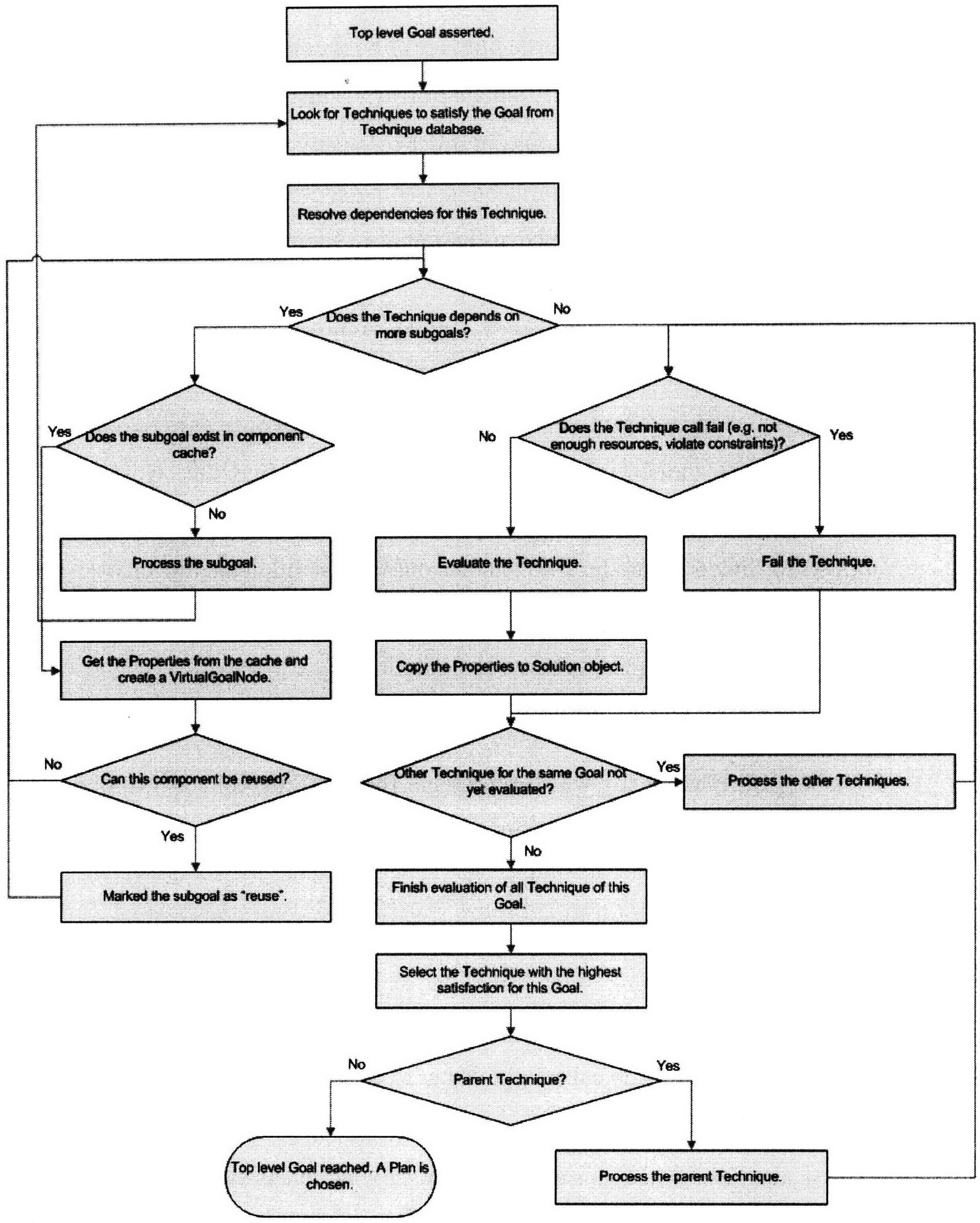

Top level Goal asserted.
Look for Techniques to satisfy the Goal from Technique database.
Resolve dependencies for this Technique.
Does the Technique depends on more subgoals?
Yes
No
Does the subgoal exist in component cache?
Yes
No
Process the subgoal.
Get the Properties from the cache and create a VirtualGoalNode.
Can this component be reused?
No
Yes
Marked the subgoal as "reuse".
Does the Technique call fail (e.g. not enough resources, violate constraints)?
No
Yes
Evaluate the Technique.
Fail the Technique.
Copy the Properties to Solution object.
Other Technique for the same Goal not yet evaluated?
Yes
Process the other Techniques.
No
Finish evaluation of all Technique of this Goal.
Select the Technique with the highest satisfaction for this Goal.
Parent Technique?
No
Yes
Top level Goal reached. A Plan is chosen.
Process the parent Technique.

Figure 3-3: The flow chart showing the Plan Tree construction and evaluation process.

semantics are added:

- To make it easy to declare a large number of hardware dependencies that are of the same type, a new construct "subgoal_array" is introduced for subgoals declaration in Techniques. For example, in order to specify a 32-bit ripple-carry adder depends on 32 full adders, declaration like "subgoal_array(goal=full_adder, number=32)" is made. An example can be found in figure 2-5.

- There are cases where several very similar architectures can share the same evaluation code, and therefore it is desirable to be able to represent them all in one single Technique script. For example, there are multiple ways to divide a 32-bit carry-select adder while they all satisfy the `Adder(bit=32)` Goal. Because the architectures are basically the same and the only difference is the parameter applied to divide the adder into smaller ones, these implementations can share the same evaluation code. To make that happen, a new "choice" phase is introduced in Technique as shown in figure 3-4.

  At the choice phase, the Technique node will fork itself, each copy with a different `divide_parameter` which represents the two different ways to divide the adder. The evaluation then continues.

As shown in the example in figure 3-4, a Technique can recursively depend on subgoal of its own kind. An ending case is needed to make sure the recursion does not loop. This is achieved through `fail()` to signal the Planner that the base case has reached. When `fail()` is called, the Planner terminates exploration of the corresponding subtree.

`fail()` is also useful in other scenarios: specifying constraints and checking the suitability of the implementation in the design. Figure 3-5 shows an adder Technique testing whether its cost violates the constraint. Figure 3-6 shows a XOR gate Technique checking whether such building block exists in the hardware environment and fails itself if not.

```
to Adder(width, carryin, db, cost_vs_delay):
    via carry_select_adder:

first:
    solution.delay = 0
    solution.cost = 0
    if goal.width < 5:
         planner.fail(``the width is less than 5'')

choice:
    self.divide_parameter = choose(choice_list=[goal.width/2, goal.width/2+1])

eval:
    self.width_one = self.divide_parameter
    self.width_two = goal.width - self.divide_parameter

subgoals:
   adder1 = Adder(width=self.width_one, carryin=goal.carryin, db=goal.db,
                  cost_vs_delay=goal.cost_vs_delay)
   adder2 = Adder(width=self.width_two, carryin=0, db=goal.db,
                  cost_vs_delay=goal.cost_vs_delay)
   adder3 = Adder(width=self.width_two, carryin=1, db=goal.db,
                  cost_vs_delay=goal.cost_vs_delay)

eval:
   solution.delay = max(subgoals.adder2.delay, subgoals.adder3.delay)
                    + subgoals.adder1.delay
   solution.cost = subgoals.adder1.cost + subgoals.adder2.cost
                    + subgoals.adder3.cost
```

Figure 3-4: The carry-select adder Technique.

```
eval:
    if solution.cost > goal.adder_max_cost:
        planner.fail(``exceed max cost!'')
```

Figure 3-5: A Technique fragment testing whether the implementation exceeds the maxinum cost.

```
eval:
    if not goal.db.query_avaliability(``XOR''):
        planner.fail(``no XOR gate avaliable!'')
```

Figure 3-6: A Technique fragment testing whether there are enough resources.

3.4.4 Goal node sharing

In order to solve the scalability problem mentioned in section 3.2, we use subgoal sharing. Subgoal sharing is to exploit the fact that there are many duplicate Goals (*e.g.* AND-gate) in the tree. As Goals represent functions needed in the circuit, the implementation choice for the same function should be identical in the same hardware environment in most cases. As a result, the Goal only needs to be evaluated once. Sharing greatly reduces the number of nodes in the tree without affecting the final decision made.

For example, figure 3-7 shows an implementation of a leftshifter. This Technique depends on an array of multiplexers. This is wasteful if the Planner expands all the multiplexer subtrees (totally 160 subtrees if width=32) in the array because they all share the same Properties in the same hardware environment, and expanding the subtrees simply takes up more memory without providing more information. To achieve Goal node sharing, the multiplexer subtree is only expanded when the Planner first meets the Goal. The expanded Goal then leaves a record in the "Goal cache" of the Planner. When the multiplexer Goal is requested the second time, the Planner looks up the Goal Properties from the "Goal cache" instead of expanding the subtree. The Goal Properties are stored in a `VirtualGoalNode` where the parent Technique can access the subgoals Properties as usual.

```
to leftshifter(width, db, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0
        self.level = int(math.log(goal.width, 2))

    subgoals:
        mux_array = subgoal_array(goal=Mux, number=self.level * goal.width,
                                  width=2, db=goal.db,
                                  cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.mux_array[0].cost * self.level * goal.width
        solution.delay = subgoals.mux_array[0].delay * self.level

    commit:
        sub_comp = {``mux2'': subgoals.mux_array[0].component}
        fa = leftshifter(``leftshifter'', component=sub_comp,
                         parameters={``width'':goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

Figure 3-7: The left shifter Technique.

3.4.5 Reusing components

Goals represent the functions required in the circuit and the Planner searches through the Techniques database to fill in the corresponding implementation. The Planner has to be careful that it does not fill in the same building block in the circuit multiple times if the component can be reused. For example, the instruction fetch stage circuit of the pipelined Beta architecture is composed of a memory to store the instructions, the D flip-flops to store the instruction address, an incrementer circuit to add 4 to the program counter, and a mux to select the next instruction. While resolving the dependencies for the memory subgoal, the Planner figures out the data memory is designed to store the instructions as well—the data memory can be *reused* as the instruction memory. Instead of expanding the subtree for the instruction memory subgoal, the Planner replaces the subgoal with a `VirtualGoalNode`, with the Properties copied the main memory Goal node. The `VirtualGoalNode` is then marked "reused" and the parent Technique can access the subgoal Properties as usual.

To support component reuse, the Planner has to know what already exists in the circuit. A data structure named the "hardware environment cache" (`env`) is created for the Planner to check and reuse the hardware components.

3.4.6 Searching and Technique selection

Heuristic search is enough for the examples so far to generate implementations that are comparable to manually optimized circuit. However, it might require more extensive search if the circuit requirements are more complex. For example, a design criteria may be "under cost x, generate the fastest implementation". In order to generate the fastest implementation, the Planner selects the fastest implementation at each decision point. However, the final design violates the cost constraint. In order to generate an acceptable design, backtracking is needed. The Planner must choose a cheaper implementation instead of the fastest one for the building blocks that are not on the critical path.

To achieve exhaustive exploration of all the possible choices, a technique called "node cloning" [4] is introduced in the Planner. The Planner tests the combination of the subgoal choices while maintaining a record of Satisfaction of all combinations. This is done by storing the Satisfaction in cloned Technique nodes that represent the subgoal choices combination. The combination with the highest Satisfaction without violating the constraint is chosen for the implementation. Figure 3-8 shows a simplified example for generating a 2-stage pipelined Beta processor. The building blocks which do not contribute to the critical path can opt for a less costly implementation after cloning to generate a feasible implementation.

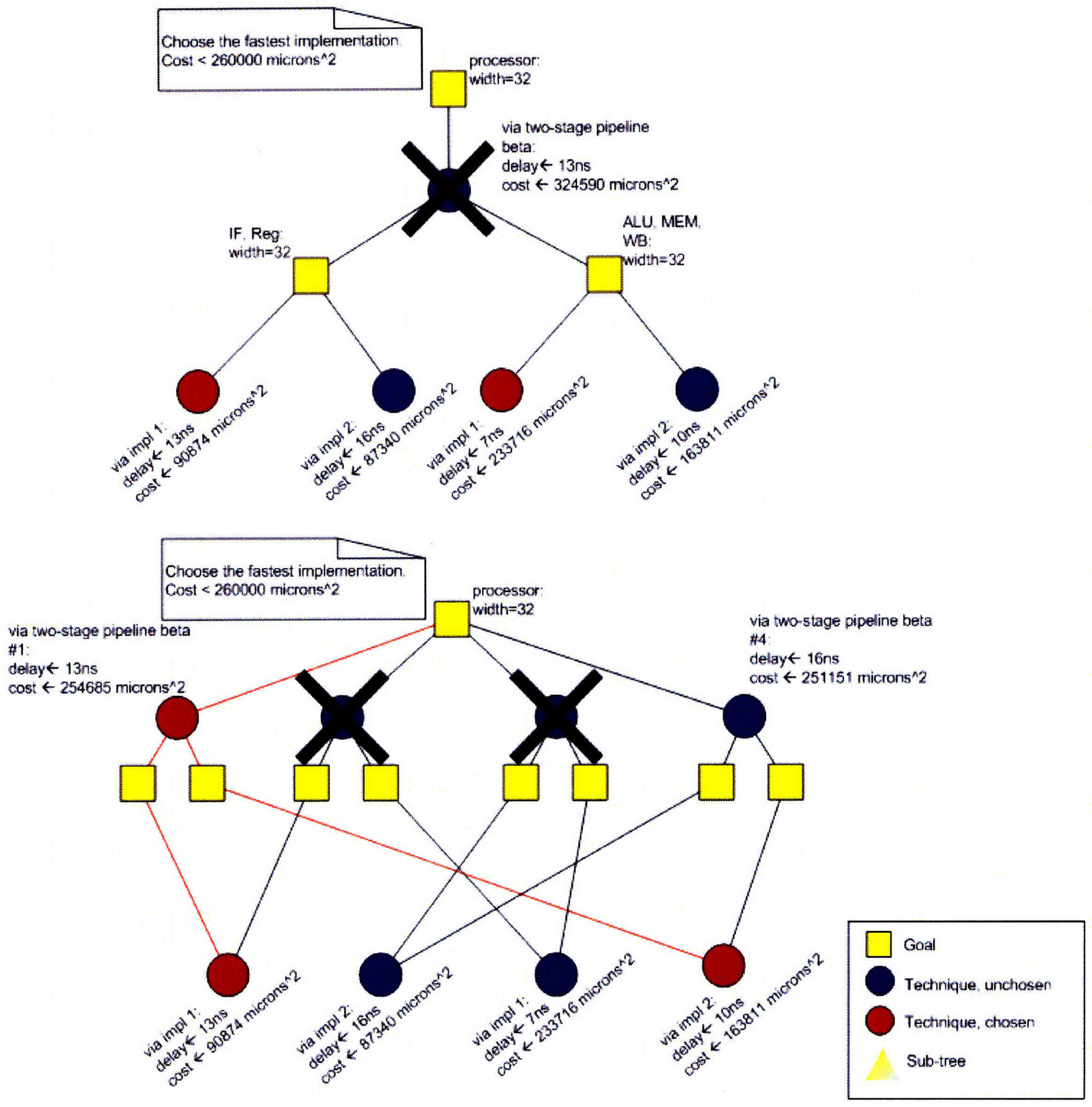


Figure 3-8: Showing how node cloning works.

Chapter 4

Application and Evaluation

4.1 Evaluation

4.1.1 Quality of the implementations generated

The implementations generated are verified using the 6.004 project checkoff file. Table 4.1 shows the qualities of the implementations generated. They are measured in terms of the area, the cycle time, and the Benmark with the corresponding points. The Benmark is the scoring system for the 6.004 design project. The smaller the circuit and the faster it completes the checkoff benchmark, the better the Benmark.

Our result shows the Planner generates good implementations under all three different decision criteria—the smallest implementation, the fastest one, and the one with the highest Benmark. The Goal specification of `processor` is shown in figure 4-1. The weighting between the cost and the delay can be modified by changing the Goal parameter `cost_vs_delay`. To choose the Benmark as the satisfaction calculation method, the Goal parameter `Benmark` is set to `True`. We include a reference Beta that represents a very highly hand-tuned design.

4.1.2 The Planner performance

For all the examples, Fide runs on a Pentium 4/1.8GHz with 512MB RAM running Linux 2.6.18. Table 4.2 shows the running time for evaluating an adder, an ALU,

```
name: processor
properties: delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False hardware_database=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Figure 4-1: The specification of the Goal processor.

Table 4.1: The quality of the implementations generated

Decision criteria	Area (*microns*^{2})	cycle time (ns)	Benmark	Points
The highest Benmark	291546	9.19	39.58	10
The fastest	322165	8.8	37.41	10
The smallest	277095	17.5	21.87	6
Best students' work	318465	9.5	35.05	10

and a Beta Plan Tree. Figure 4-2 shows the running time for evaluating an ALU Plan Tree of different widths and it consists of two graphs. One includes the carry select adder Technique and the other does not. The running time of graph *withCSA* increases exponentially while the adder width increases exponentially. This is because the number of nodes in the Tree increases proportionally with the width as the carry select architecture recursively breaks up the adder into smaller ones. The smaller adder Goals expand into subtrees, with size proportional to the width.

On the other hand, the running time of graph *withoutCSA* increases at a slower rate while the width increases. Unlike graph *withCSA* which includes the carry select adder Technique, increasing the width does not introduce new Goal node type (smaller adder) into the Plan Tree. Even though the width is increasing, many building blocks (*e.g.* full adders, multiplexers) are duplicate. Because of the Goal node sharing mechanism, increasing the width only increases the number of `VirtualGoalNode` to represent the duplicate building blocks, in which the `VirtualGoalNode` does not expand.

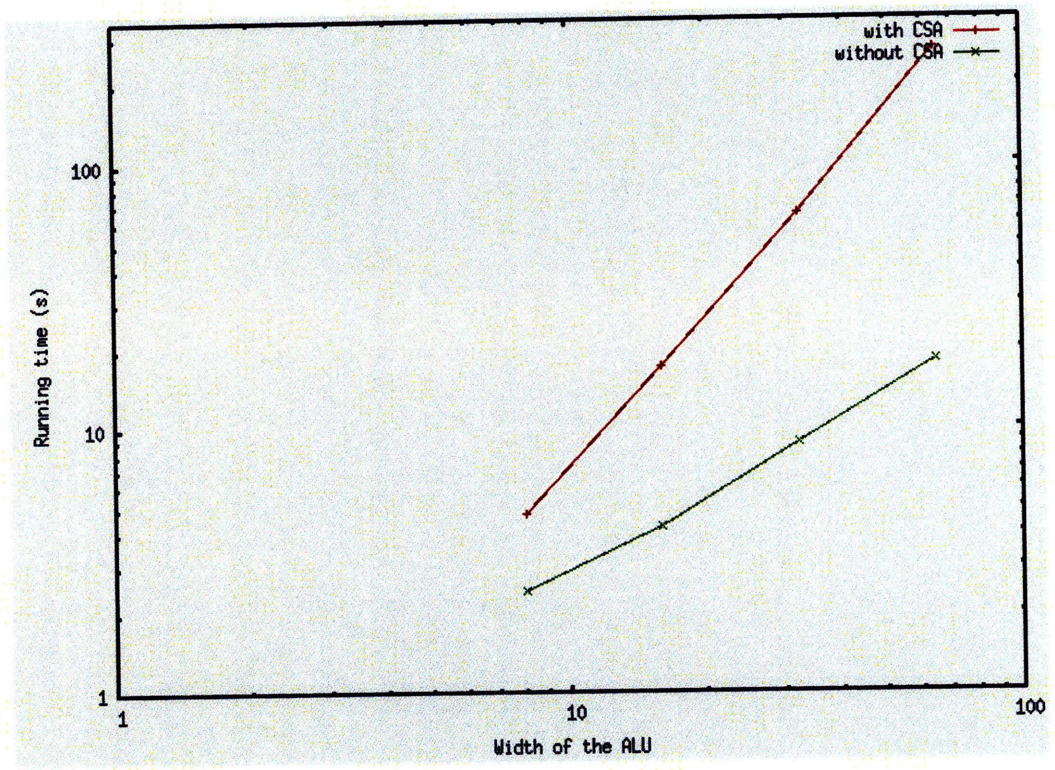


Figure 4-2: The graph showing the running time for evaluating an ALU Plan Tree of different widths.

Table 4.2: Time needed to evaluate different circuits

A Beta (width=32)	An ALU (width=32)	An Adder (width=32)
134.68s	64.51s	59.58s

Chapter 5

Implementation

Given the system design and architecture presented in the chapter 3, this chapter details the system interface and the implementation of Fide. Fide is implemented in Python2.5 and the GUI is implemented using the wxPython2.8 library. Even though no code of the Planner depends on special feature unique to Python, using an interpreted language make implementing Fide easier. This is to ease parsing and executing the Goal satisfaction formula and the Technique evaluation scripts on-the-fly.

5.1 The Planner API

The Planner API can be divided into three types: the Plan Tree API, the GUI API and the User API.

5.1.1 The Plan Tree API

The Plan Tree API is the interfaces for manipulating the Plan Tree. It includes the following:

- `plan(goal, scheduler_class)` This starts the construction of the Plan Tree. It is given the top level Goal instance. The top level Goal instance is generated by the Goal Parser by parsing the Goal specification. Instead of the default value stated in the specification, the parser modifies the Goal Parameters according to the users' input. The `scheduler_class` is a scheduler instance that the Planner uses to queue the tree nodes for evaluation.

  During `plan()`, the Planner constructs the Plan Tree by finding Techniques that satisfy the top level Goal, and recursively matching the subgoals of each Technique it finds.

- `evaluate()` During `evaluate()`, the Planner polls the scheduler for tree nodes for evaluation. If it is a Goal node, evaluation means computing the Satisfaction values for each child Technique and choosing the one with the highest Satisfaction. If it is a Technique node, evaluation means executing the evaluation phases to compute the Goal Proeprties.

  When the users are modifying the design, the affected Tree nodes are marked as `invalid`. The users call `evaluate()` to invoke re-evaluation of the Plan Tree, where all the invalid nodes are enqueued by the scheduler.

- `commit()` When `commit()` is called, the Planner recursively calls commit on the chosen Technique nodes, where it stops at the leaves. The Technique nodes pass the building blocks they instantiate up the Tree to contruct the implementation.

5.1.2 The GUI API

The GUI API is the interface where the Planner provides Plan Tree information for the GUI for display purpose. It includes the following:

- `add_inspector(inspector)` This adds the GUI handler to the Planner, which the Planner invokes whenever there are changes in the Plan Tree. The handler should provide the following callbacks:

  - `finish_evaluation(plan)` This is called after evaluation is done.

  - `finish_commit(plan)` This is called after commit is done.

For both callbacks, the Plan object is passed. The Plan provides a pointer to the top level Goal, and a method to create a snapshot of the Plan Tree. The snapshot is useful for the GUI to create the tree view.

5.1.3 The User API

The User API is the interface where the users modify the Plan Tree. It includes the following:

- `modify(goal)` This is useful for changing the Goal Parameters. The caller creates a new Goal instance according to the new Goal Parameters and passes it to `modify`. The Planner changes the top level Goal using the new Goal instance and passes the changed Parameters down the tree during re-evaluation.

- `update_goal(goal_name, goal)` This is useful for changing the Satisfaction formula of Goals. The caller creates a new Goal instance using the new Satisfaction formula. Based on the goal name given, the Planner updates the corresponding Goal node during re-evaluation using the new Goal instance.

- `add_technique(goal_name, technique_class)` This is useful for adding a new implementation recipe on-the-fly. The caller creates the Technique class using the Technique interpreter and passed it to this method. Before re-evaluation, the Planner creates a new Technique node and attaches it to the corresponding parent Goal. The new Technique node is marked as "invalid", which will be enqueued and evaluated.

- `update_technique(goal_name, technique_class)` This is useful for updating an implementation recipe on-the-fly. The update mechanism is mostly the same as adding a Technique, except that the Planner is replacing the old Technique node with the new one.

5.2 The Planner GUI architecture

Figure 5-1 shows the architecture of the GUI which consists of the followings:

1. *Plan Tree snapshot.* The Plan Tree snapshot is given by the Planner, which reflects the latest Plan chosen. Based on the snapshot, the GUI generates the tree view.

2. *Model.* Based on users' input, the Model sends signals to modify the Plan Tree through the Planner inspector. After re-evalaution, it signals View to re-draw the Plan Tree. The users commit the Plan similarly through Model.

   It is also responsible for "manipulating" the GUI internal Plan Tree representation *e.g.* searching the tree, tracing the Technique dependencies *etc.*

3. *View.* It draws the GUI: the tree view with all the highlightings and display options, as well as the information/commit/statistics/Technique script display panels.

4. *Controller.* It interprets the users' input and signals the responsible callbacks for further actions. For example, right-clicking on a virtual Goal node in the tree view means highlighting the original Goal node as pointed by the virtual node.

5. *Planner inspector.* It acts as a "broker" between the Planner and the Model. It can handle multiple Plan Trees at the same time. Each Plan Tree corresponds to a seperate thread. Thus, the evaluation of multiple Plan Trees can take place in parallel while the users can still interact with the GUI. After evaluation/commit, it posts an event back to the Model to signal update for the latest changes. It also generates the Plan Tree internal representation based on the Plan Tree snapshot.

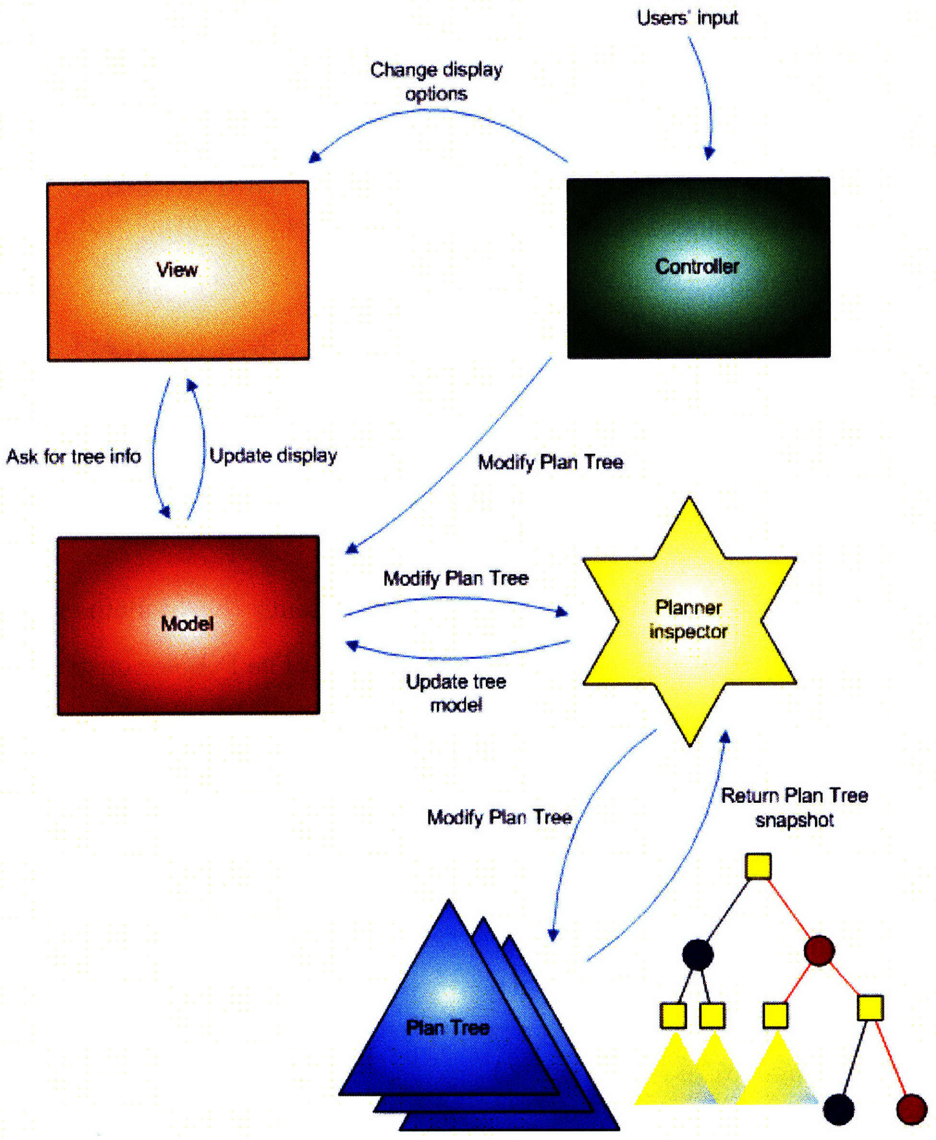


Figure 5-1: The Planner GUI architecture.

5.3 The JSim Component library

The building blocks in the JSim Component library are divided into three data types: Pebble, Composite and Circuit. In addition, the data type named Connector provides the building blocks the notion of input and output connection to other components. A Connector of width=1 represents a wire connection, and with width greater than 1 represents a bus connection. Table 5.1 shows the constructor API for these four data types. The library provides a block diagram abstraction to describe hardware. This is to avoid exposing unnecessary internal structure of the circuit in Technique to ease Technique programming.

Table 5.1: Constructor API for the JSim Component Library

Pebble	`Pebble(function, input=[], output=[])`
Composite	`Composite(function, components=[], parameters={})`
Circuit	`Circuit(composite)`
Connector	`Connector(name, width)`

Pebble is the most primitive type, which represents the fundamental building blocks in the circuit that do not have any dependency. The Pebble constructor simply takes the function name and the Connectors for the input/output ports. Figure 5-2 shows how a Pebble is used. The leaf Technique instantiates an instance of Pebble in the commit phase for the logic gate it represents. After commit, the Pebble instance is copied up the tree in `solution.component` for the parent Techniques to composite the circuit using this gate.

```
commit:
    in1 = Connector(name=''in1'')
    in2 = Connector(name=''in2'')
    in3 = Connector(name=''in3'')
    out = Connector(name=''out'')
    solution.component = Pebble(''nor3'', input=[in1, in2, in3], output=[out])
```

Figure 5-2: Instantiate a Pebble in Technique.

A Composite represents a collection of interconnected components (Pebble or Composites). The programmer creates a new type of building block that is composed

```
commit:
    sub_comp = {''mux4'': subgoals.mux_array[0].component,
                ''adder'': subgoals.add.component,
                ''boole'': subgoals.boole.component,
                ''shift'': subgoals.shift.component,
                ''compare'': subgoals.compare.component}
    alu = ALU(''alu'', component=sub_comp, parameters={''width'':goal.width})
    unmatch = alu.initialize()
    if unmatch:
        planner.fail(''Commit error: interface not match!'')
    solution.component = alu.instantiate()
```

Figure 5-3: Instantiate a Composite in Technique.

of smaller components by subclassing Composite. The programmer defines the interface (*i.e.* input and output ports) and how the inner components are connected during subclassing. Section 5.3 details how to subclass Composite. Figure 5-3 shows how a Composite is used. The Technique copies the sub-components from its subgoals and passed them to `ALU`, a Composite subclass. The Technique also parameterizes `ALU` by passing it the Goal Parameter, *i.e.* the data width. It calls `initialize()` to connect the sub-components. During `initialize()`, `ALU` also checks whether the interface of the sub-components match and fails the Technique otherwise. After creating the circuit using the sub-components, the Technique instantiates the ALU and passes it up the tree for composition of even more complex building blocks.

A Circuit represents the actual instantiation of the building blocks in the implementation. Pebble and Composite create the circuit definitions and Circuit "instantiates" the building blocks by creating the netlists. The Circuit Constructor is given instances of Composites. By subclassing Circuit, the programmer defines how the connections to the outside world are made for the different Composites. Figure 5-4 shows how a Circuit is used. The circuit definition of Beta, which is a Composite subclass instance copied from the subgoal, is passed to the Circuit constructor `Beta_processor`. The Connectors to the outside world are passed and the netlist of `Beta_processor` is created. Finally, `output_circuit()` creates the JSim implementation of Beta, along with all the circuit definitions of the sub-components.

```
commit:
    beta = subgoals.beta.component
    beta_circuit = beta_processor(beta)
    clk = Connector(name=''clk'')
    reset = Connector(name=''reset'')
    qid = Connector(name=''id'', width=32)
    mrd = Connector(name=''mrd'', width=32)
    ia = Connector(name=''ia'', width=32)
    ma = Connector(name=''ma'', width=32)
    moe = Connector(name=''moe'')
    wr = Connector(name=''wr'')
    werf = Connector(name=''werf'')
    mwd = Connector(name=''mwd'', width=32)
    beta_circuit.create(clk=clk,
                        reset=reset,
                        qid=qid,
                        mrd=mrd,
                        ia=ia,
                        ma=ma,
                        moe=moe,
                        wr=wr,
                        werf=werf,
                        mwd=mwd)
    solution.component = output_circuit(beta_circuit)
```

Figure 5-4: Instantiate a Circuit in Technique.

5.3.1 Composite

The following shows the internal methods of Composite. They are for connecting the interal components while extending `initialize()`:

1. `_init_interface()` It creates the interface of the Composite using the input/output Connectors instantiated in the constructor.

2. `_connect_point(a, b)` It connects the two given points a and b.

3. `_connect(*args)` It connects all the points in `*args` together.

4. `_connect_component(comp, input=[], output=[])` It connects the given internal component using the Connectors in the `input` and `output` array.

5. `_connect_memory(width, nlocations, readPort=[], writePort=[], contents='')` It creates a memory and connects it using the Connectors given in the `readPort` and `writePort` array. `readPort` and `writePort` are arrays of read/write interfaces, where each interface is an array of Connectors in the order of `[oe, clk, wen, addr, data]`. `oe` is the output enable, `clk` is the clock, `wen` is the write enable, `addr` is the address with the most significant bit first, and `data` is the data inputs/tristate outputs.

5.3.2 Connector

The followings show the API of a Connector:

1. `bit(*args)` It returns a subset of the bus. If `*args` only consists of 1 parameter, it means selecting the particular bit. Otherwise, the first and second parameters are the first and second indices of the bus sequence.

2. `num(num)` It returns multiple copies of the same connection.

Both methods return a new Connector representing the new type of connection. Thus, the following can be made to describe duplicating a bus subset:

```
new_connector = data.bit(15,0).num(2)
```

Figure 5-5 shows how to implement the ripple carry adder by subclassing Composite and using Connectors.

```
class Ripple_Carry_Adder(JSimComponent):
    def __init__(self, name, component=[], parameters={}):

        self.width = parameters[''width'']
        self.bit = self.width-1
        self.input = [Connector(name=''ALUFN0''),
                      Connector(name=''A'', width=self.width),
                      Connector(name=''B'', width=self.width)]
        self.output = [Connector(name=''S'', width=self.width),
                      Connector(name=''z''),
                      Connector(name=''v''),
                      Connector(name=''n'')]
        JSimComponent.__init__(self, name, component, parameters)
        self.interface = self._initInterface()


    def initialize(self):
        bx = Connector(''bx'', width=self.width)
        c = Connector(''c'', width=self.width)
        self._connect_component(self.component[''xor2''],
                                input=[self.B, self.ALUFN0.num(self.width)],
                                output=[bx])
        self._connect_component(self.component[''fulladder''],
                                input=[self.A.bit(0), bx.bit(0), self.ALUFN0],
                                output=[c.bit(0), self.S.bit(0)])
        self._connect_component(self.component[''fulladder''],
                                input=[self.A.bit(self.bit, 1),
                                       bx.bit(self.bit, 1),
                                       c.bit(self.bit-1, 0)],
                                output=[c.bit(self.bit, 1),
                                        self.S.bit(self.bit, 1)])
        self._connect_component(self.component[''z_circuit''],
                                input=[self.S],
                                output=[self.z])
        self._connect(self.S.bit(self.bit), self.n)
        self._connect_component(self.component[''v_circuit''],
                                input=[self.A.bit(self.bit), bx.bit(self.bit),
                                       self.S.bit(self.bit)],
                                output=[self.v])
```

Figure 5-5: Implementing the ripple carry adder using the JSim component library.

Chapter 6

Related Work

6.1 High-level hardware description languages

The building blocks in Fide are implemented using existing languages and are represented as Techniques for evaluation. Besides popular choices like Verilog or VHDL, there are options of using general-purpose high-level programming language to describe hardware. There are C-based languages like HandelC[8], OCAPI[9], HardwareC[10] and Transmogrifier-C[11]. Recently there is MyHDL[12]—a Python-based hardware description language. These projects have two common goals: (1.) Using a familiar, high-level imperative language to ease hardware programming. (2.) enhancing software/hardware co-design by providing a unified design environment. Some other languages have special design rationale behind, *e.g.* Pebble[13], which is tailored for describing reconfigurable hardware. Functional programming also inspires hardware design. There are Haskell-based languages like Bluespec[14] and Lava[15]. Bluespec enhances design of concurrent system by managing the complexity of shared resources with special compilation technology. Lava enables design to be simulated, verified and instantiated using the same high-level description. Finally, there are Prolog inspired languages *e.g.* [16] is designed to ease design verification.

In Fide, using an existing language allows the implementation to enjoy whatever language features or compiler technologies that come with the language. Some languages provide special features to allow modeling hardware at different abstraction levels, or a high-level parametrized and reusable hardware library to enhance programming. Some of them come with compilation advantages: able to synthesize and verify the circuit efficiently and systematically, or the compiler is optimized for different applications to give better synthesis result. Some languages enhance software/hardware co-design that it is easy to represent, transform and partition the system into hardware/software synthesizable descriptions.

6.2 Circuit optimization and AI Planning

Fide does not compete with existing hardware optimization algorithms [17]. Instead, Fide complements them. Goal-oriented programming encodes the decision logic, and how the Planner makes the decision can be customized—according to any optimization method. In other words, what the Planner fundamentally offers is a data structure i.e. the Plan tree that represents the decision logic. The Plan tree enumerates all known strategies for the implementation, where a path from the root Goal node to leaf Techniques represents a particular implementation strategy [4]. This also distinguishes our "planning" from planning in AI sense. AI Planning is a much broader area involving research like how to model a problem, how to program the planning algorithm, how to execute the sequence of actions *etc.* While Fide is not considered to be in the AI discipline, a lot of related work *e.g.* dependencies backtracking could inspire Fide on how to improve the "smartness" of the Planner in decision making.

6.3 Hardware design methodologies

There is work focusing on design rationale for different architectures. For example, Lin, et al. [18] focus on design method for pipelined heterogeneous multiprocessor system (to be specific, ASIP), Todman, et al. [19] focus on reconfigurable architecture (FPGAs) and Pham, et al. [20] focus on multi-core architecture. Knowing when to apply the right design strategy, or combine a hybrid of them to give optimal design, is difficult and highly application specific. Goals provide an abstraction layer above all these design methods. Each design method is an alternative to satisfy the Goals and is evaluated and compared systematically. There is also work that focuses on automatic circuit exploration. For example, Verma, et al. [21] focus on the automatic optimization of arithmetic circuits. All of these are considered as an implementation choice which can be easily added in Fide for evaluation under the open-ended decision logic nature of Goal-oriented programming.

Chapter 7

Conclusion

7.1 Future work

7.1.1 More efficient search

In the current Planner, exhaustive search is needed to give optimal design, which is a very expensive operation in terms of both computation and memory. There are two ways to improve this:

- To be able to perform smart partial search. The hints could be inferred from past decision history, cleverly collected user preferences, or from the design itself (*e.g.* only perform exhaustive search on components on the critical path).
- Because the Plan tree is essentially an AND-OR tree, it could be possibly translated into a SAT problem, which can be solved by a SAT solver. The transformation could be assisted with avaliable modeling languages like Alloy [22].

7.1.2 Better estimate of the circuit properties

The circuit properties computed in Techniques are all programmed manually. This is not practical, not feasible for huge design and not accurate as it does not include the routing cost *etc.* It is desirable to invoke external libraries to perform more realistic circuit synthesis. Theoretically, this can be done as the Technique eval phase can be any arbitrary code. Thus, more engineering effort *e.g.* explore which libraries to use, understand the synthesis result and translate that into Goal Properties *etc.* are needed to make it happen.

7.2 Conclusion

This thesis presents Fide—applying Goal-oriented programming to enhance hardware design. Goals represent the decision points in the hardware implementation decision logic and Techniques represent the recipes to implement the circuit. By exploiting the explicit seperations of the decision logic and the implementation chocies, the Planner is able to generate an implementation of the target circuit automatically. Users' preferences can be added to generate circuits for different scenarios: for different hardware environments, under different circuit constraints, or different implementation criteria *etc.* A Beta processor is implemented using Fide. The qualities of the implementation is comparable and a logic minimization to those optimized manually.

Appendix A

The Planner GUI

Before the users can specify the refinements, they need a thorough understanding of the properties of the current design. Fide has a Planner Viewer that allows the user to explore the Plan Tree and current design.

1. The Plan Tree is presented as a collapsable tree to ease browsing. The tree nodes are colored for easy identification. The chosen Techniques are in blue, the failing Techniques with fail reasons are in red, the `VirtualGoalNode` are in grey, and the remaining nodes in black. The reused components are marked as "reused".

2. The users can choose to display the Goal Properties, Goal Parameters, Satisfaction values and Satisfaction formula on the tree node.

3. A search tree control is provided to quickly find any node in a huge tree.

4. The subgoals in an array, which are of the same type, are represented as one node with the array size shown to save space and memory.

5. By right-clicking the `VirtualGoalNode` (shared Goals/reused components), the Viewer highlights the "original" Goals where the `VirtualGoalNode` copies the Goal Properties from. The users can then retrieve the subtree information of the "original" Goal. In the other way round, by selecting "show reuse" on the "original" Goal, all parent Techniques of the related `VirtualGoalNodes` are

colored orange. This is to ease verifying whether Goal sharing is appropriate for each parent Technique.

6. To get a quick understanding of the Plan chosen, the users can choose to show the chosen nodes only.
7. The users can view multiple Plan Trees side-by-side. This is useful for comparing the same circuit with different constraints/requirements, or taking references from other designs *etc.*
8. It has a statistics panel listing all the Goals with the number of occurance.
9. By choosing "show Technique code" for the highlighted Technique, the Viewer displays the corresponding Technique script.
10. It has a commit panel showing the implementation generated.
11. It includes user interfaces for modifying the design.

The followings are the screenshots of the GUI.

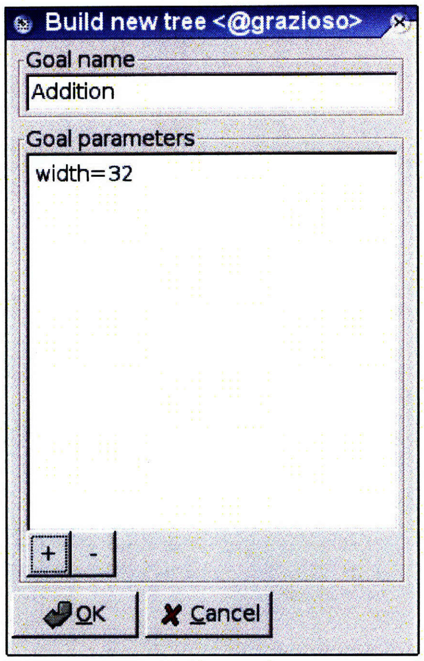
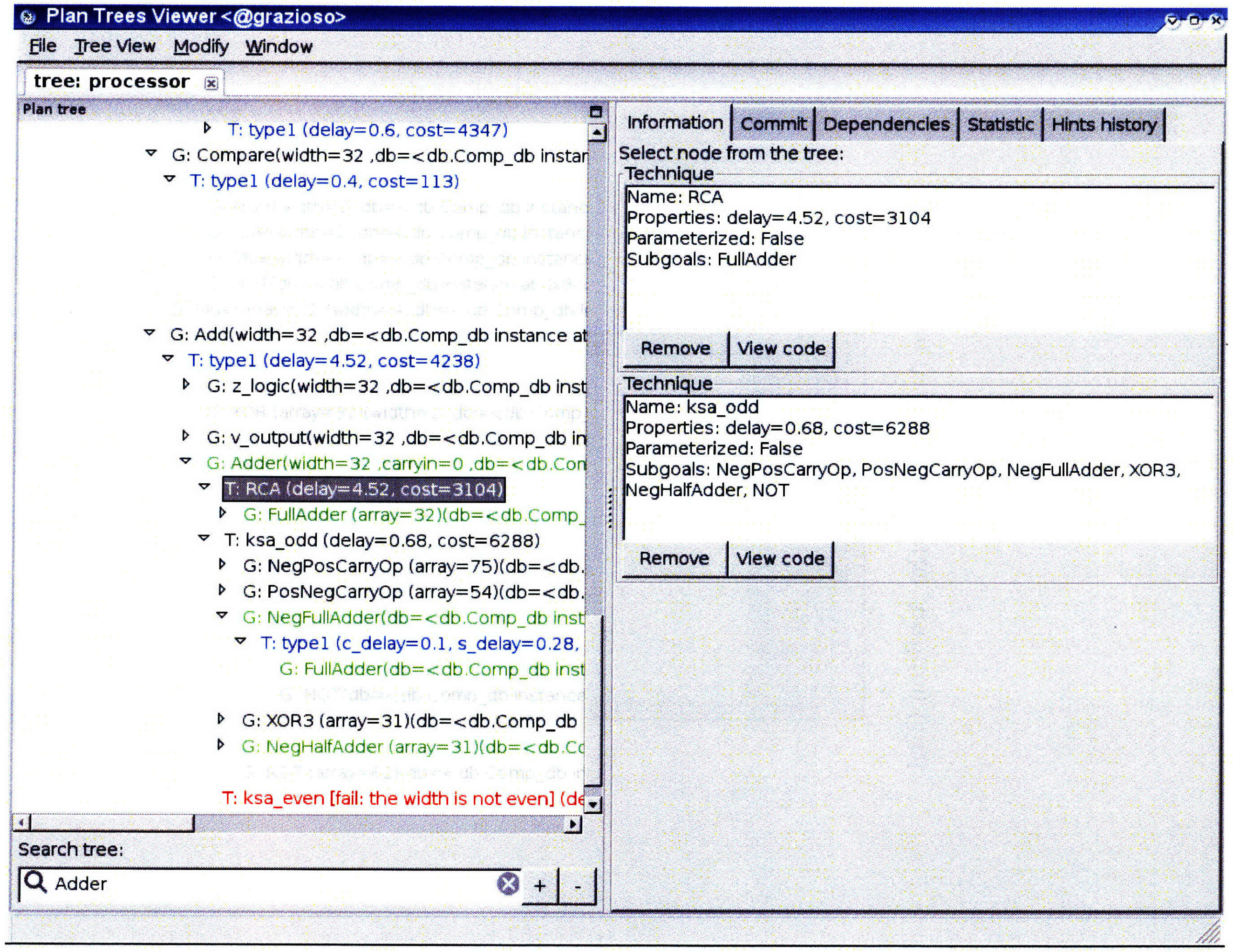


Figure A-1: Asserting the Addition Goal.


Plan Trees Viewer <@grazioso>
File Tree View Modify Window
tree: processor
Plan tree
Information Commit Dependencies Statistic Hints history
Remove
View code
Search tree:
+
-

```
T: type1 (delay=0.6, cost=4347)
G: Compare(width=32 ,db=<db.Comp_db instar
  T: type1 (delay=0.4, cost=113)
G: Add(width=32 ,db=<db.Comp_db instance at
  T: type1 (delay=4.52, cost=4238)
    G: z_logic(width=32 ,db=<db.Comp_db inst
    G: v_output(width=32 ,db=<db.Comp_db in
    G: Adder(width=32 ,carryin=0 ,db=<db.Con
      T: RCA (delay=4.52, cost=3104)
        G: FullAdder (array=32)(db=<db.Comp_
      T: ksa_odd (delay=0.68, cost=6288)
        G: NegPosCarryOp (array=75)(db=<db.
        G: PosNegCarryOp (array=54)(db=<db.
        G: NegFullAdder(db=<db.Comp_db inst
          T: type1 (c_delay=0.1, s_delay=0.28,
            G: FullAdder(db=<db.Comp_db inst
        G: XOR3 (array=31)(db=<db.Comp_db
        G: NegHalfAdder (array=31)(db=<db.Co
      T: ksa_even [fail: the width is not even] (de
```

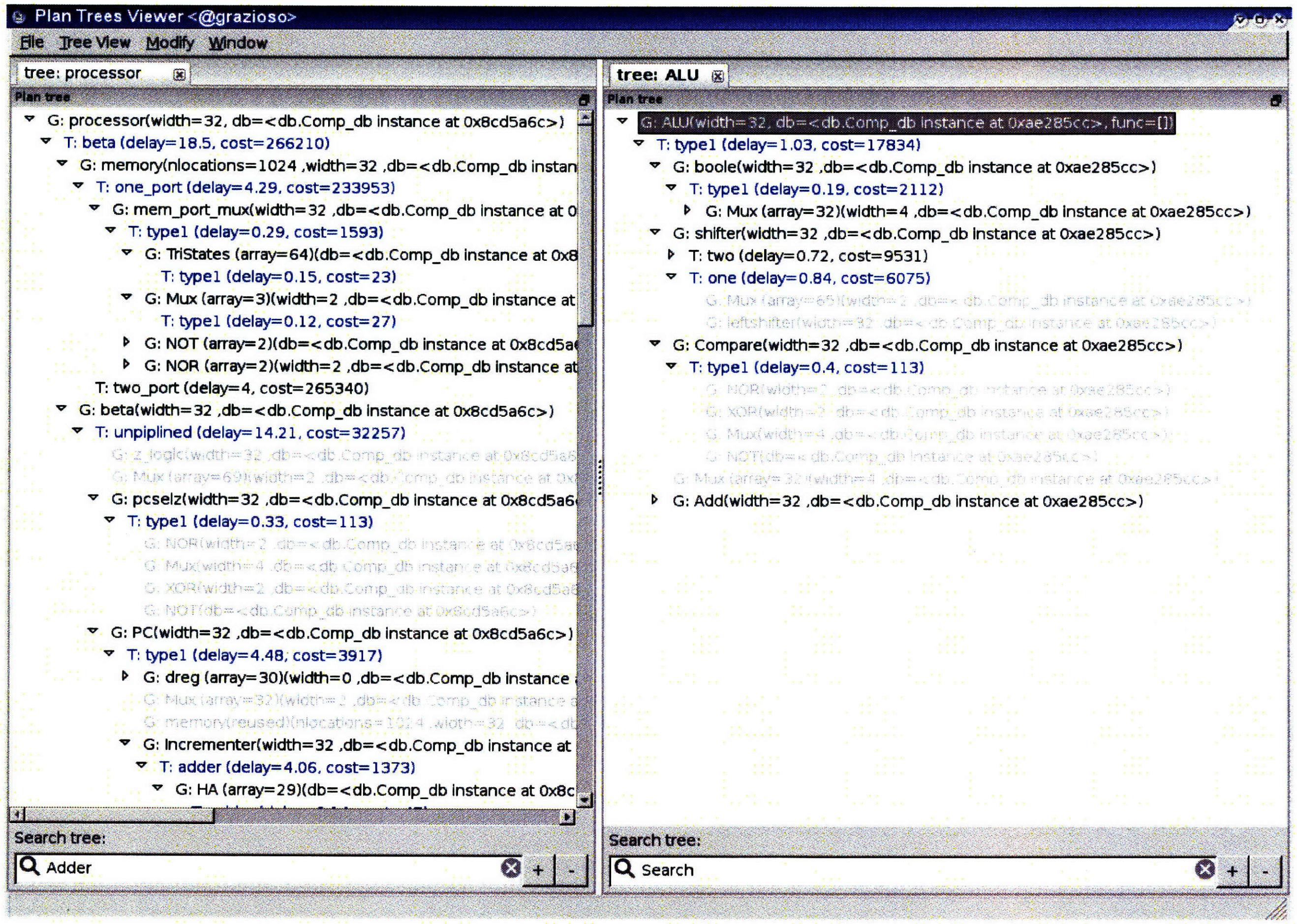
Search tree: Adder

Select node from the tree:

```
Technique
Name: RCA
Properties: delay=4.52, cost=3104
Parameterized: False
Subgoals: FullAdder

Technique
Name: ksa_odd
Properties: delay=0.68, cost=6288
Parameterized: False
Subgoals: NegPosCarryOp, PosNegCarryOp, NegFullAdder, XOR3,
NegHalfAdder, NOT
```

Figure A-2: The information panel showing the nodes properties.



```
tree: processor
Plan tree
▼ G: processor(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)
  ▼ T: beta (delay=18.5, cost=266210)
    ▼ G: memory(nlocations=1024 ,width=32 ,db=<db.Comp_db instan
      ▼ T: one_port (delay=4.29, cost=233953)
        ▼ G: mem_port_mux(width=32 ,db=<db.Comp_db instance at 0
          ▼ T: type1 (delay=0.29, cost=1593)
            ▼ G: TriStates (array=64)(db=<db.Comp_db instance at 0x8
                T: type1 (delay=0.15, cost=23)
            ▼ G: Mux (array=3)(width=2 ,db=<db.Comp_db instance at
                T: type1 (delay=0.12, cost=27)
            ▶ G: NOT (array=2)(db=<db.Comp_db instance at 0x8cd5a
            ▶ G: NOR (array=2)(width=2 ,db=<db.Comp_db instance at
      T: two_port (delay=4, cost=265340)
    ▼ G: beta(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
      ▼ T: unpiplined (delay=14.21, cost=32257)
          G: z_logic(width=32 ,db=<db.Comp_db instance at 0x8cd5a6
          G: Mux (array=69)(width=2 ,db=<db.Comp_db instance at 0x
        ▼ G: pcselz(width=32 ,db=<db.Comp_db instance at 0x8cd5a6
          ▼ T: type1 (delay=0.33, cost=113)
              G: NOR(width=2 ,db=<db.Comp_db instance at 0x8cd5a
              G: Mux(width=4 ,db=<db.Comp_db instance at 0x8cd5a6
              G: XOR(width=2 ,db=<db.Comp_db instance at 0x8cd5a6
              G: NOT(db=<db.Comp_db instance at 0x8cd5a6c>)
        ▼ G: PC(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
          ▼ T: type1 (delay=4.48, cost=3917)
            ▶ G: dreg (array=30)(width=0 ,db=<db.Comp_db instance
              G: Mux (array=32)(width=2 ,db=<db.Comp_db instance
              G: memory(reused)(nlocations=1024 ,width=32 ,db=<db
            ▼ G: Incrementer(width=32 ,db=<db.Comp_db instance at
              ▼ T: adder (delay=4.06, cost=1373)
                ▼ G: HA (array=29)(db=<db.Comp_db instance at 0x8c
Search tree:
Adder
```

```
tree: ALU
Plan tree
▼ G: ALU(width=32, db=<db.Comp_db instance at 0xae285cc>, func=[])
  ▼ T: type1 (delay=1.03, cost=17834)
    ▼ G: boole(width=32 ,db=<db.Comp_db instance at 0xae285cc>)
      ▼ T: type1 (delay=0.19, cost=2112)
        ▶ G: Mux (array=32)(width=4 ,db=<db.Comp_db instance at 0xae285cc>)
    ▼ G: shifter(width=32 ,db=<db.Comp_db instance at 0xae285cc>)
      ▶ T: two (delay=0.72, cost=9531)
      ▼ T: one (delay=0.84, cost=6075)
          G: Mux (array=65)(width=2 ,db=<db.Comp_db instance at 0xae285cc>)
          G: leftshifter(width=32 ,db=<db.Comp_db instance at 0xae285cc>)
    ▼ G: Compare(width=32 ,db=<db.Comp_db instance at 0xae285cc>)
      ▼ T: type1 (delay=0.4, cost=113)
          G: NOR(width=2 ,db=<db.Comp_db instance at 0xae285cc>)
          G: XOR(width=2 ,db=<db.Comp_db instance at 0xae285cc>)
          G: Mux(width=4 ,db=<db.Comp_db instance at 0xae285cc>)
          G: NOT(db=<db.Comp_db instance at 0xae285cc>)
      G: Mux (array=32)(width=4 ,db=<db.Comp_db instance at 0xae285cc>)
    ▶ G: Add(width=32 ,db=<db.Comp_db instance at 0xae285cc>)
Search tree:
Search
```

Figure A-3: Viewing two Plan Trees side-by-side.

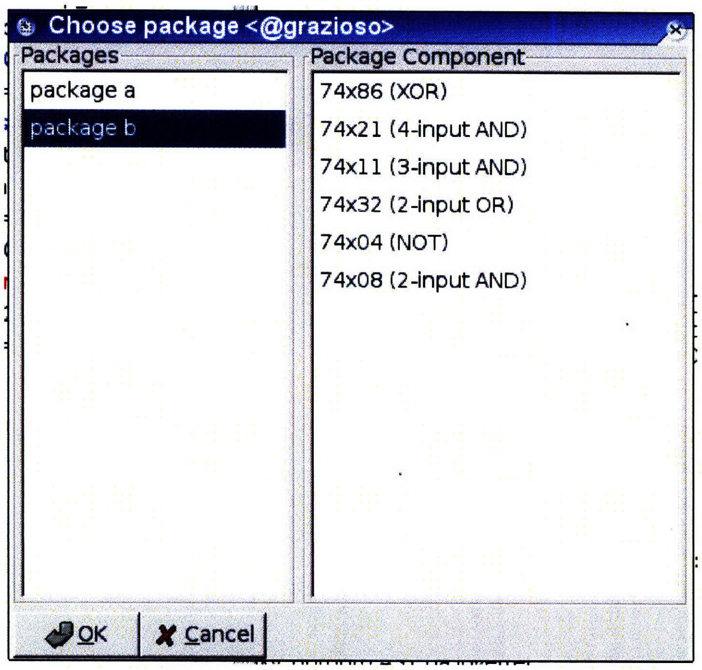


Figure A-4: Choosing the hardware package.

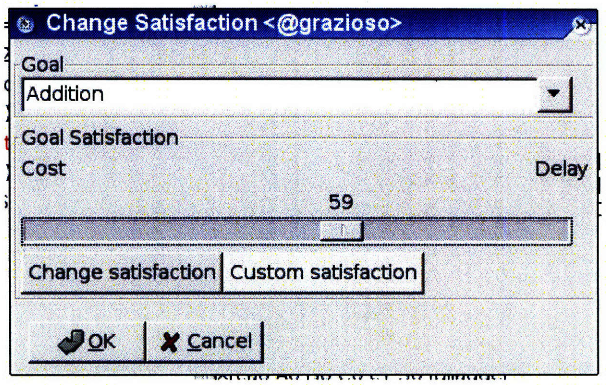
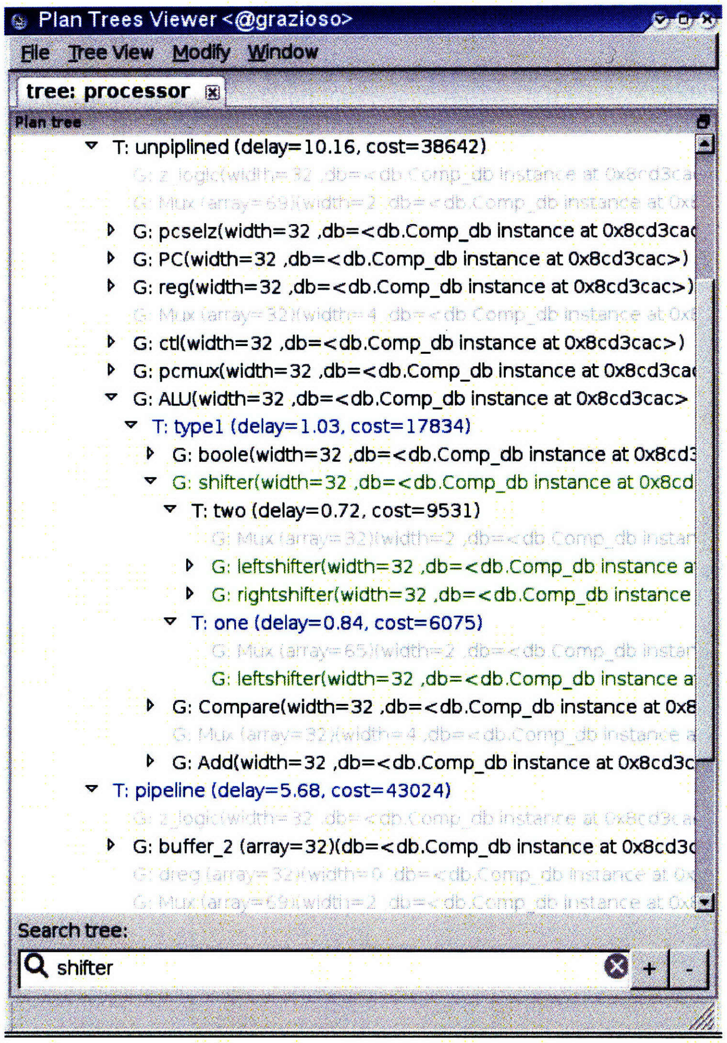
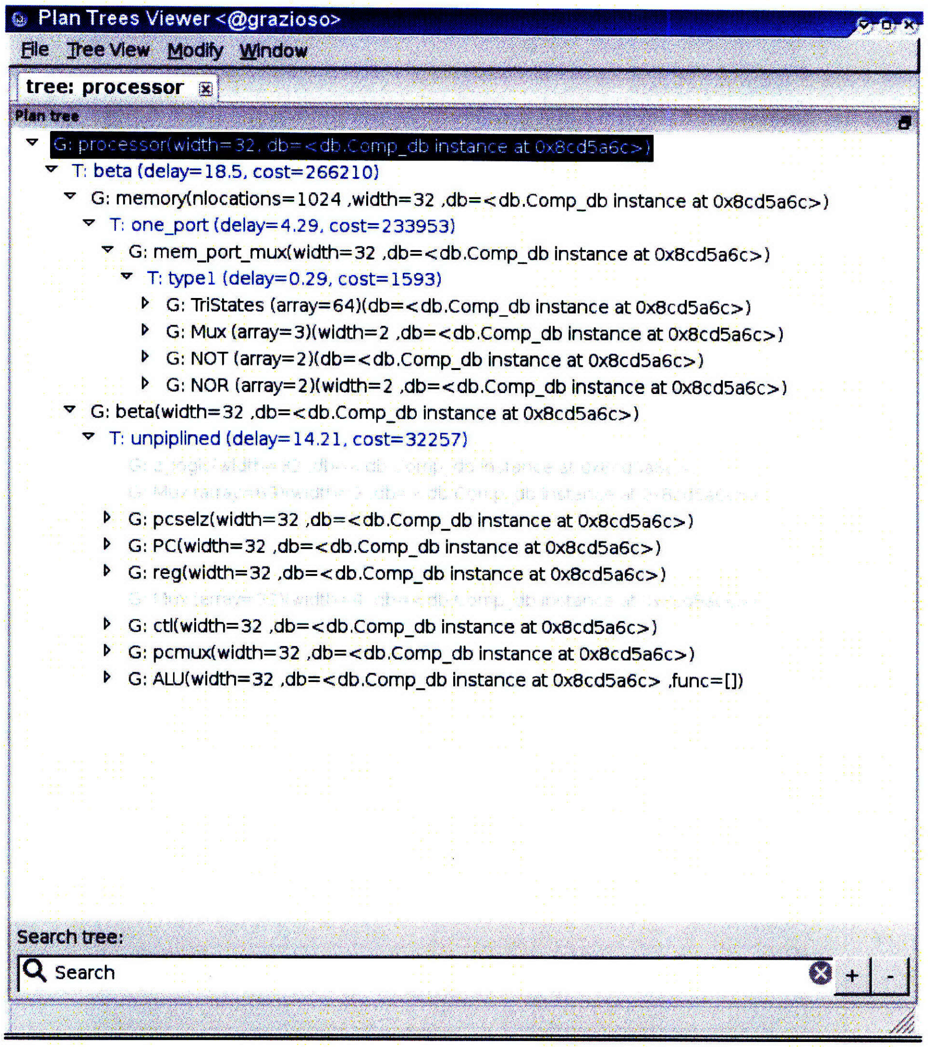


Figure A-5: Changing the satisfaction formula.



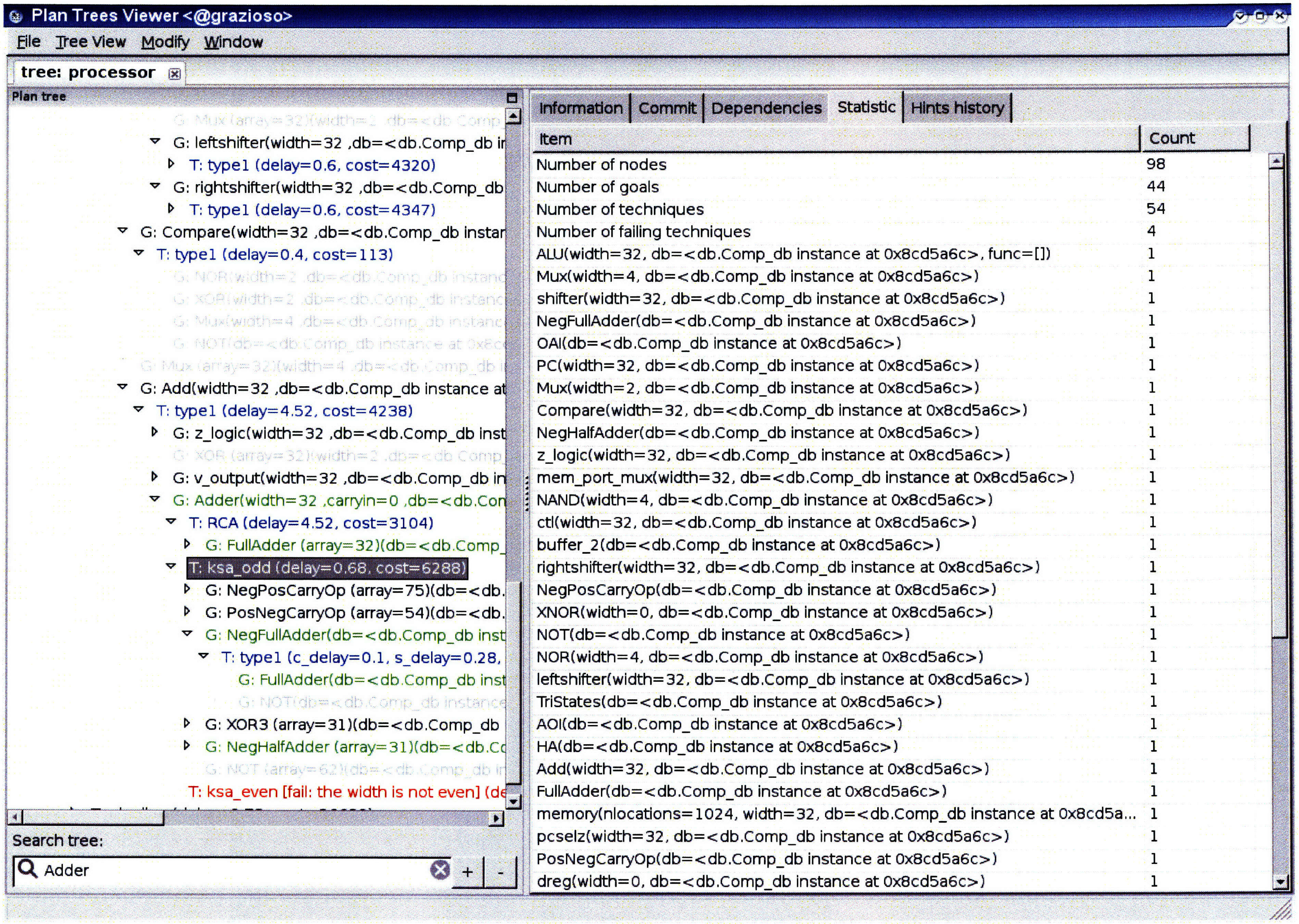
```
T: unpiplined (delay=10.16, cost=38642)
  G: z_logic(width=32 ,db=<db.Comp_db instance at 0x8cd3ca
  G: Mux (array=69)(width=2 ,db=<db.Comp_db instance at 0x
  G: pcselz(width=32 ,db=<db.Comp_db instance at 0x8cd3cac
  G: PC(width=32 ,db=<db.Comp_db instance at 0x8cd3cac>)
  G: reg(width=32 ,db=<db.Comp_db instance at 0x8cd3cac>)
  G: Mux (array=32)(width=4 ,db=<db.Comp_db instance at 0x
  G: ctl(width=32 ,db=<db.Comp_db instance at 0x8cd3cac>)
  G: pcmux(width=32 ,db=<db.Comp_db instance at 0x8cd3cac
  G: ALU(width=32 ,db=<db.Comp_db instance at 0x8cd3cac>
    T: type1 (delay=1.03, cost=17834)
      G: boole(width=32 ,db=<db.Comp_db instance at 0x8cd3
      G: shifter(width=32 ,db=<db.Comp_db instance at 0x8cd
        T: two (delay=0.72, cost=9531)
          G: Mux (array=32)(width=2 ,db=<db.Comp_db instan
          G: leftshifter(width=32 ,db=<db.Comp_db instance a
          G: rightshifter(width=32 ,db=<db.Comp_db instance
        T: one (delay=0.84, cost=6075)
          G: Mux (array=65)(width=2 ,db=<db.Comp_db instan
          G: leftshifter(width=32 ,db=<db.Comp_db instance a
      G: Compare(width=32 ,db=<db.Comp_db instance at 0x8
      G: Mux (array=32)(width=4 ,db=<db.Comp_db instance a
      G: Add(width=32 ,db=<db.Comp_db instance at 0x8cd3c
T: pipeline (delay=5.68, cost=43024)
  G: z_logic(width=32 ,db=<db.Comp_db instance at 0x8cd3ca
  G: buffer_2 (array=32)(db=<db.Comp_db instance at 0x8cd3c
  G: dreg (array=32)(width=0 ,db=<db.Comp_db instance at 0x
  G: Mux (array=69)(width=2 ,db=<db.Comp_db instance at 0x
```

Figure A-6: Searching the tree, the search results are colored green.



```
G: processor(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)
  T: beta (delay=18.5, cost=266210)
    G: memory(nlocations=1024 ,width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
      T: one_port (delay=4.29, cost=233953)
        G: mem_port_mux(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
          T: type1 (delay=0.29, cost=1593)
            G: TriStates (array=64)(db=<db.Comp_db instance at 0x8cd5a6c>)
            G: Mux (array=3)(width=2 ,db=<db.Comp_db instance at 0x8cd5a6c>)
            G: NOT (array=2)(db=<db.Comp_db instance at 0x8cd5a6c>)
            G: NOR (array=2)(width=2 ,db=<db.Comp_db instance at 0x8cd5a6c>)
    G: beta(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
      T: unpiplined (delay=14.21, cost=32257)
        [illegible]
        [illegible]
        G: pcselz(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: PC(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: reg(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        [illegible]
        G: ctl(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: pcmux(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: ALU(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c> ,func=[])
```

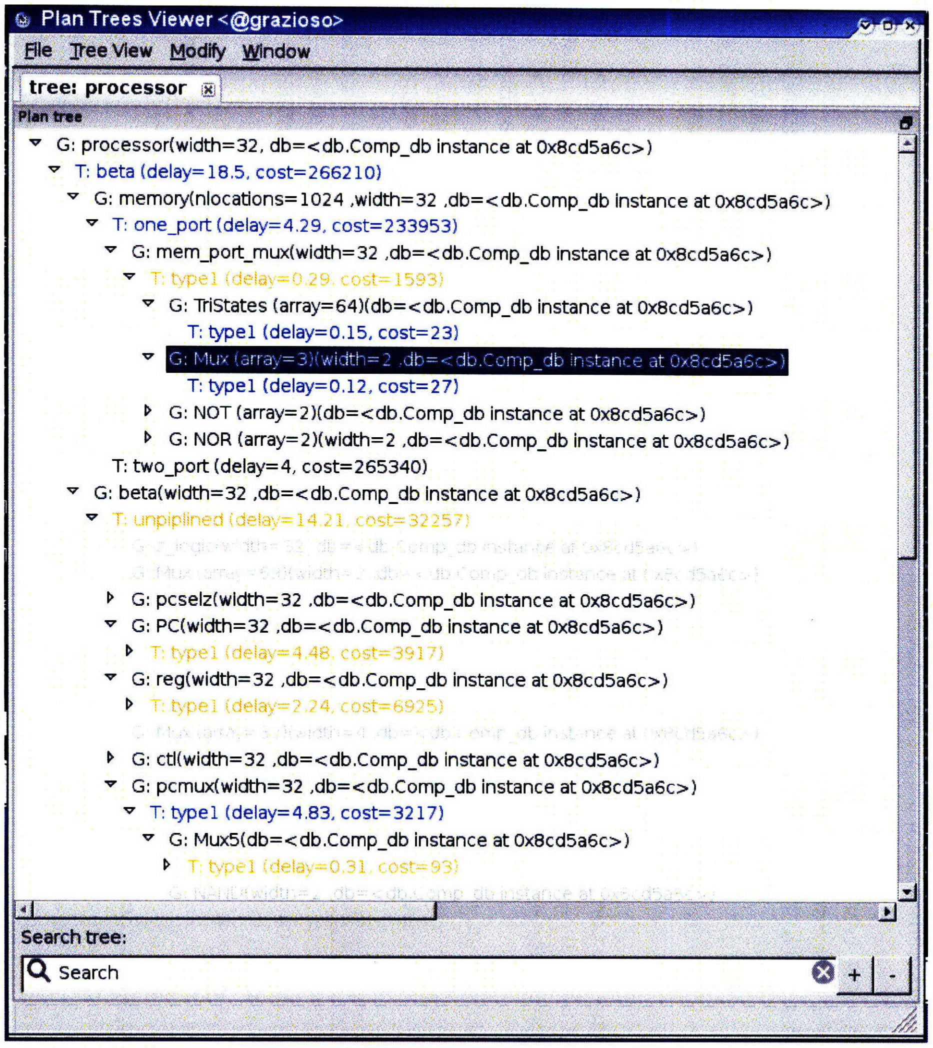
Figure A-7: Displaying the chosen nodes only to ease browsing the chosen Plan.



```
            G: Mux (array=32)(width=2 ,db=<db.Comp
        G: leftshifter(width=32 ,db=<db.Comp_db ir
            T: type1 (delay=0.6, cost=4320)
        G: rightshifter(width=32 ,db=<db.Comp_db
            T: type1 (delay=0.6, cost=4347)
    G: Compare(width=32 ,db=<db.Comp_db instar
        T: type1 (delay=0.4, cost=113)
            G: NOR(width=2 ,db=<db.Comp_db instanc
            G: XOR(width=2 ,db=<db.Comp_db instanc
            G: Mux(width=4 ,db=<db.Comp_db instanc
            G: NOT(db=<db.Comp_db instance at 0x8c
    G: Mux (array=32)(width=4 ,db=<db.Comp_db i
    G: Add(width=32 ,db=<db.Comp_db instance at
        T: type1 (delay=4.52, cost=4238)
            G: z_logic(width=32 ,db=<db.Comp_db inst
            G: XOR (array=32)(width=2 ,db=<db.Comp
            G: v_output(width=32 ,db=<db.Comp_db in
            G: Adder(width=32 ,carryin=0 ,db=<db.Con
                T: RCA (delay=4.52, cost=3104)
                    G: FullAdder (array=32)(db=<db.Comp_
                T: ksa_odd (delay=0.68, cost=6288)
                    G: NegPosCarryOp (array=75)(db=<db.
                    G: PosNegCarryOp (array=54)(db=<db.
                    G: NegFullAdder(db=<db.Comp_db inst
                        T: type1 (c_delay=0.1, s_delay=0.28,
                            G: FullAdder(db=<db.Comp_db inst
                            G: NOT(db=<db.Comp_db instance
                    G: XOR3 (array=31)(db=<db.Comp_db
                    G: NegHalfAdder (array=31)(db=<db.Co
                    G: NOT (array=62)(db=<db.Comp_db in
                T: ksa_even [fail: the width is not even] (de
```

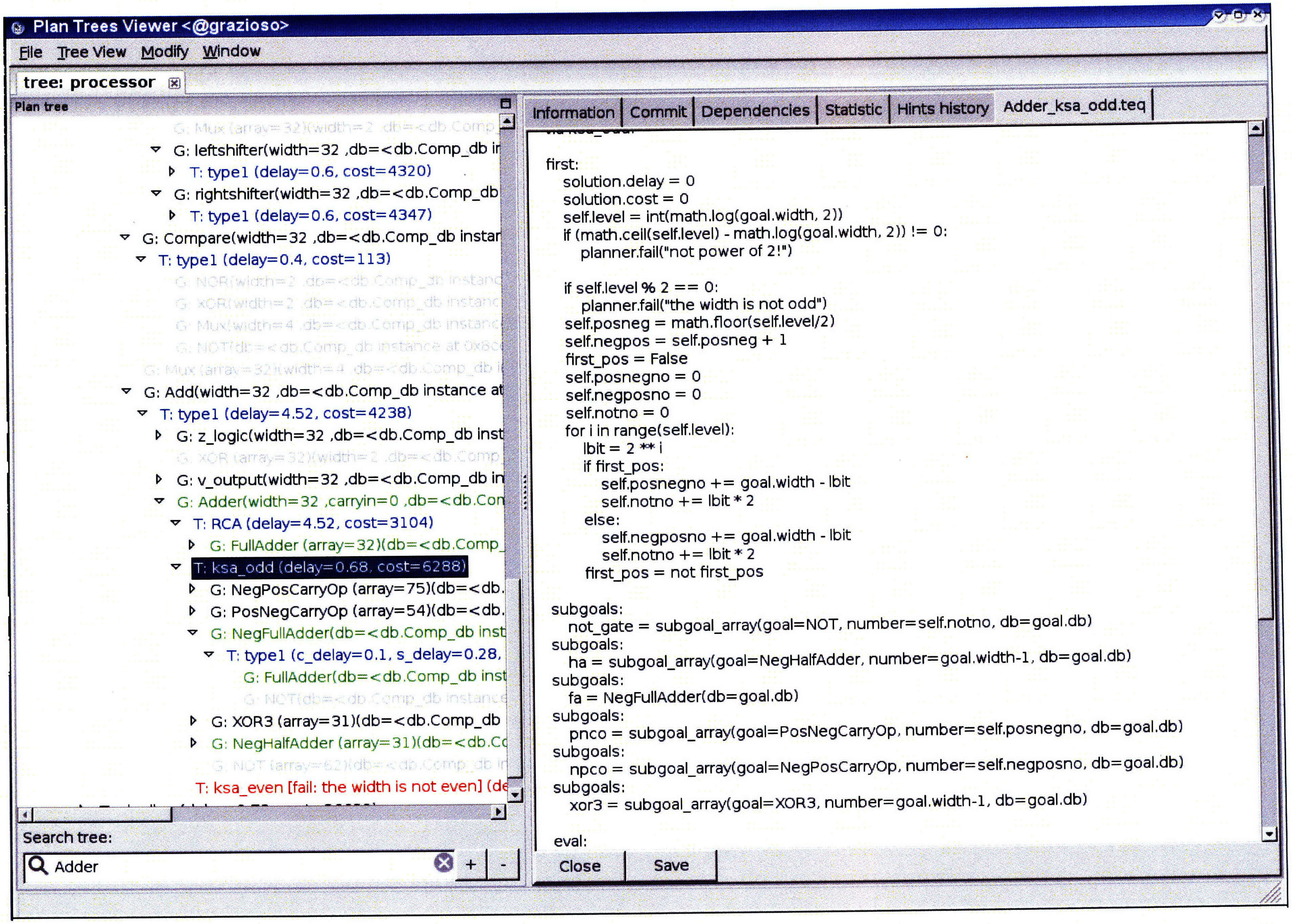
Item	Count
Number of nodes	98
Number of goals	44
Number of techniques	54
Number of failing techniques	4
ALU(width=32, db=<db.Comp_db instance at 0x8cd5a6c>, func=[])	1
Mux(width=4, db=<db.Comp_db instance at 0x8cd5a6c>)	1
shifter(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
NegFullAdder(db=<db.Comp_db instance at 0x8cd5a6c>)	1
OAI(db=<db.Comp_db instance at 0x8cd5a6c>)	1
PC(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
Mux(width=2, db=<db.Comp_db instance at 0x8cd5a6c>)	1
Compare(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
NegHalfAdder(db=<db.Comp_db instance at 0x8cd5a6c>)	1
z_logic(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
mem_port_mux(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
NAND(width=4, db=<db.Comp_db instance at 0x8cd5a6c>)	1
ctl(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
buffer_2(db=<db.Comp_db instance at 0x8cd5a6c>)	1
rightshifter(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
NegPosCarryOp(db=<db.Comp_db instance at 0x8cd5a6c>)	1
XNOR(width=0, db=<db.Comp_db instance at 0x8cd5a6c>)	1
NOT(db=<db.Comp_db instance at 0x8cd5a6c>)	1
NOR(width=4, db=<db.Comp_db instance at 0x8cd5a6c>)	1
leftshifter(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
TriStates(db=<db.Comp_db instance at 0x8cd5a6c>)	1
AOI(db=<db.Comp_db instance at 0x8cd5a6c>)	1
HA(db=<db.Comp_db instance at 0x8cd5a6c>)	1
Add(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
FullAdder(db=<db.Comp_db instance at 0x8cd5a6c>)	1
memory(nlocations=1024, width=32, db=<db.Comp_db instance at 0x8cd5a...	1
pcselz(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)	1
PosNegCarryOp(db=<db.Comp_db instance at 0x8cd5a6c>)	1
dreg(width=0, db=<db.Comp_db instance at 0x8cd5a6c>)	1

Figure A-8: The statistics panel listing all the Goals with the number of occurance.



```
G: processor(width=32, db=<db.Comp_db instance at 0x8cd5a6c>)
  T: beta (delay=18.5, cost=266210)
    G: memory(nlocations=1024 ,width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
      T: one_port (delay=4.29, cost=233953)
        G: mem_port_mux(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
          T: type1 (delay=0.29, cost=1593)
            G: TriStates (array=64)(db=<db.Comp_db instance at 0x8cd5a6c>)
              T: type1 (delay=0.15, cost=23)
            G: Mux (array=3)(width=2 ,db=<db.Comp_db instance at 0x8cd5a6c>)
              T: type1 (delay=0.12, cost=27)
            G: NOT (array=2)(db=<db.Comp_db instance at 0x8cd5a6c>)
            G: NOR (array=2)(width=2 ,db=<db.Comp_db instance at 0x8cd5a6c>)
      T: two_port (delay=4, cost=265340)
    G: beta(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
      T: unpiplined (delay=14.21, cost=32257)
        G: z_logic(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: Mux (array=5)(width=2 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: pcselz(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: PC(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
          T: type1 (delay=4.48, cost=3917)
        G: reg(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
          T: type1 (delay=2.24, cost=6925)
        G: Mux (array=5)(width=4 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: ctl(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
        G: pcmux(width=32 ,db=<db.Comp_db instance at 0x8cd5a6c>)
          T: type1 (delay=4.83, cost=3217)
            G: Mux5(db=<db.Comp_db instance at 0x8cd5a6c>)
              T: type1 (delay=0.31, cost=93)
                G: NAND(width=2 ,db=<db.Comp_db instance at 0x8cd5a6c>)
```

Figure A-9: The `VirtualGoalNodes` in grey, the original Goal node highlighted and the parent Techniques of the sharing nodes in orange.



Plan Trees Viewer <@grazioso>
File Tree View Modify Window
tree: processor
Plan tree
Information Commit Dependencies Statistic Hints history Adder_ksa_odd.teq
Search tree:
Adder
+ -
Close Save

```
G: Mux (array=32)(width=2 ,db=<db.Comp_
  G: leftshifter(width=32 ,db=<db.Comp_db in
    T: type1 (delay=0.6, cost=4320)
  G: rightshifter(width=32 ,db=<db.Comp_db
    T: type1 (delay=0.6, cost=4347)
G: Compare(width=32 ,db=<db.Comp_db instar
  T: type1 (delay=0.4, cost=113)
    G: NOR(width=2 ,db=<db.Comp_db instanc
    G: XOR(width=2 ,db=<db.Comp_db instanc
    G: Mux(width=4 ,db=<db.Comp_db instanc
    G: NOT(db=<db.Comp_db instance at 0x8c
G: Mux (array=32)(width=4 ,db=<db.Comp_db i
G: Add(width=32 ,db=<db.Comp_db instance at
  T: type1 (delay=4.52, cost=4238)
    G: z_logic(width=32 ,db=<db.Comp_db inst
    G: XOR (array=32)(width=2 ,db=<db.Comp_
    G: v_output(width=32 ,db=<db.Comp_db in
    G: Adder(width=32 ,carryin=0 ,db=<db.Con
      T: RCA (delay=4.52, cost=3104)
        G: FullAdder (array=32)(db=<db.Comp_
      T: ksa_odd (delay=0.68, cost=6288)
        G: NegPosCarryOp (array=75)(db=<db.
        G: PosNegCarryOp (array=54)(db=<db.
        G: NegFullAdder(db=<db.Comp_db inst
          T: type1 (c_delay=0.1, s_delay=0.28,
            G: FullAdder(db=<db.Comp_db inst
            G: NOT(db=<db.Comp_db instance
        G: XOR3 (array=31)(db=<db.Comp_db
        G: NegHalfAdder (array=31)(db=<db.Co
        G: NOT (array=62)(db=<db.Comp_db in
      T: ksa_even [fail: the width is not even] (de
```

```
first:
    solution.delay = 0
    solution.cost = 0
    self.level = int(math.log(goal.width, 2))
    if (math.ceil(self.level) - math.log(goal.width, 2)) != 0:
        planner.fail("not power of 2!")

    if self.level % 2 == 0:
        planner.fail("the width is not odd")
    self.posneg = math.floor(self.level/2)
    self.negpos = self.posneg + 1
    first_pos = False
    self.posnegno = 0
    self.negposno = 0
    self.notno = 0
    for i in range(self.level):
        lbit = 2 ** i
        if first_pos:
            self.posnegno += goal.width - lbit
            self.notno += lbit * 2
        else:
            self.negposno += goal.width - lbit
            self.notno += lbit * 2
        first_pos = not first_pos

subgoals:
    not_gate = subgoal_array(goal=NOT, number=self.notno, db=goal.db)
subgoals:
    ha = subgoal_array(goal=NegHalfAdder, number=goal.width-1, db=goal.db)
subgoals:
    fa = NegFullAdder(db=goal.db)
subgoals:
    pnco = subgoal_array(goal=PosNegCarryOp, number=self.posnegno, db=goal.db)
subgoals:
    npco = subgoal_array(goal=NegPosCarryOp, number=self.negposno, db=goal.db)
subgoals:
    xor3 = subgoal_array(goal=XOR3, number=goal.width-1, db=goal.db)

eval:
```

Figure A-10: Showing the Technique script.

Appendix B

Goal Specifications and Technique Scripts

Adder.goal

```
name:Adder
properties:delay cost
functional attributes: width=0 pin=[] carryin=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Adder_RCA.teq

```
from jsim_test import adderRCA
to Adder(width, pin, carryin, db, Benmark, cost_vs_delay):
    via RCA:

    first:
        solution.delay = 0
        solution.cost = 0
        if goal.width < 5:
            planner.fail("ripple carry adder fails: width < 5")

    subgoals:
        fas = subgoal_array(goal=FullAdder, number=goal.width, db=goal.db,
                    cost_vs_delay=goal.cost_vs_delay,
                            Benmark=goal.Benmark)
    eval:
        for fa in subgoals.fas:
           solution.delay += fa.s_delay

        solution.cost += subgoals.fas[0].cost * goal.width


    commit:
        sub_comp = {"fulladder": subgoals.fas[0].component}
        fa = adderRCA("rca", component=sub_comp, parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

Adder_ksa_even.teq

```
from jsim_test import koggeStoneAdder
import math
to Adder(width, pin, carryin, cost_vs_delay, Benmark, db):
    via ksa_even:

    first:
        solution.delay = 0
        solution.cost = 0
        self.level = int(math.log(goal.width, 2))
        if (math.ceil(self.level) - math.log(goal.width, 2)) != 0:
            planner.fail("not power of 2!")
        if self.level % 2 == 1:
            planner.fail("log(width) is not even")
        self.posneg = self.level/2
        self.negpos = self.level/2 #odd + 1
        first_pos = True
        self.posnegno = 0
        self.negposno = 0
        self.notno = 0
        for i in range(self.level):
            lbit = 2 ** i
            if first_pos:
                self.posnegno += goal.width - lbit
                self.notno += lbit * 2
            else:
                self.negposno += goal.width - lbit
                self.notno += lbit * 2
            first_pos = not first_pos

    subgoals:
        not_gate = subgoal_array(goal=gate_fix_width, number=self.notno,
                                 func="not", db=goal.db,
                                 cost_vs_delay=goal.cost_vs_delay,
                                  Benmark=goal.Benmark)
    subgoals:
        ha = subgoal_array(goal=HalfAdder, number=goal.width-1, db=goal.db,
                           cost_vs_delay=goal.cost_vs_delay,
                           Benmark=goal.Benmark)
    subgoals:
        fa = FullAdder(db=goal.db, cost_vs_delay=goal.cost_vs_delay,
                       Benmark=goal.Benmark)
    subgoals:
        pnco = subgoal_array(goal=PosNegCarryOp, number=self.posnegno, db=goal.db,
                             cost_vs_delay=goal.cost_vs_delay,
                             Benmark=goal.Benmark)
    subgoals:
        npco = subgoal_array(goal=NegPosCarryOp, number=self.negposno, db=goal.db,
                             cost_vs_delay=goal.cost_vs_delay,
                             Benmark=goal.Benmark)
    subgoals:
        xor3 = subgoal_array(goal=XOR3, number=goal.width-1, db=goal.db,
                             cost_vs_delay=goal.cost_vs_delay,
                             Benmark=goal.Benmark)

    eval:
        solution.cost = subgoals.ha[0].cost * (goal.width-1) + \
                        subgoals.fa.cost + subgoals.pnco[0].cost * self.posnegno + \
                        subgoals.npco[0].cost * self.negposno + \
                        subgoals.not_gate[0].cost * \
                        self.notno + subgoals.xor3[0].cost * (goal.width-1)
        solution.delay = subgoals.ha[0].delay + subgoals.pnco[0].delay * self.posneg\
                         + subgoals.npco[0].delay * self.negpos + subgoals.xor3[0].delay

    commit:
        sub_comp = {"halfadder": subgoals.ha[0].component,
                    "fulladder": subgoals.fa.component,
                    "posnegcarryop": subgoals.pnco[0].component,
                    "negposcarryop": subgoals.npco[0].component,
                    "xor3": subgoals.xor3[0].component,
                    "inverter": subgoals.not_gate[0].component}
        fa = koggeStoneAdder("sub_adder", component=sub_comp, parameters={"width":32})
        fa.initialize()
        solution.component = fa.instantiate()
```

Adder_ksa_odd.teq

```
from jsim_test import koggeStoneAdder
import math
to Adder(width, pin, carryin, cost_vs_delay, Benmark, db):
    via ksa_odd:
```

```
first:
    solution.delay = 0
    solution.cost = 0
    self.level = int(math.log(goal.width, 2))
    if (math.ceil(self.level) - math.log(goal.width, 2)) != 0:
        planner.fail("not power of 2!")

    if self.level % 2 == 0:
        planner.fail("log(width) is not odd")
    self.posneg = math.floor(self.level/2)
    self.negpos = self.posneg + 1
    first_pos = False
    self.posnegno = 0
    self.negposno = 0
    self.notno = 0
    for i in range(self.level):
        lbit = 2 ** i
        if first_pos:
            self.posnegno += goal.width - lbit
            self.notno += lbit * 2
        else:
            self.negposno += goal.width - lbit
            self.notno += lbit * 2
        first_pos = not first_pos

subgoals:
    not_gate = subgoal_array(goal=gate_fix_width, number=self.notno,
                             func="not", db=goal.db,
                             cost_vs_delay=goal.cost_vs_delay,
                             Benmark=goal.Benmark)
subgoals:
    ha = subgoal_array(goal=NegHalfAdder, number=goal.width-1, db=goal.db,
                       cost_vs_delay=goal.cost_vs_delay,
                       Benmark=goal.Benmark)
subgoals:
    fa = NegFullAdder(db=goal.db,
                      cost_vs_delay=goal.cost_vs_delay,
                      Benmark=goal.Benmark)
subgoals:
    pnco = subgoal_array(goal=PosNegCarryOp, number=self.posnegno, db=goal.db,
                         cost_vs_delay=goal.cost_vs_delay,
                         Benmark=goal.Benmark)
subgoals:
    npco = subgoal_array(goal=NegPosCarryOp, number=self.negposno, db=goal.db,
                         cost_vs_delay=goal.cost_vs_delay,
                         Benmark=goal.Benmark)
    subgoals:
        xor3 = subgoal_array(goal=XOR3, number=goal.width-1, db=goal.db,
                             cost_vs_delay=goal.cost_vs_delay,
                             Benmark=goal.Benmark)

    eval:
        solution.cost = subgoals.ha[0].cost * (goal.width-1) + subgoals.fa.cost + \
                        subgoals.pnco[0].cost * self.posnegno + subgoals.npco[0].cost * \
                        self.negposno + subgoals.not_gate[0].cost * self.notno + \
                        subgoals.xor3[0].cost * (goal.width-1)
        solution.delay = subgoals.ha[0].delay + subgoals.pnco[0].delay * self.posneg\
                         + subgoals.npco[0].delay * self.negpos  + subgoals.xor3[0].delay

    commit:
        sub_comp = {"neghalfadder": subgoals.ha[0].component,
                    "negfulladder": subgoals.fa.component,
                    "posnegcarryop": subgoals.pnco[0].component,
                    "negposcarryop": subgoals.npco[0].component,
                    "xor3": subgoals.xor3[0].component,
                    "inverter": subgoals.not_gate[0].component}
        fa = koggeStoneAdder("sub_adder", component=sub_comp, parameters={"width":32})
        fa.initialize()
        solution.component = fa.instantiate()
```

Adder_CSA.teq

```
from add_cost import adding_cost
from jsim_test import adderRCA_CSAlo, adderRCA_CSAhi, \
adder_CSAlo, adder_CSAhi, adder_CSAhihi

def compute_breakpoint(width, number, goal):

    result = []
    for i in range(number):

        bp = goal.width * 1/2 + i
        pin_set = goal.pin[:bp]
        result.append([bp, pin_set])
    return result

to Adder(width, pin, carryin, db, cost_vs_delay, Benmark):
    via CSA:

    first:
        solution.delay = 0
        solution.cost = 0
```

```
choice:
    self.break_choice = choose(choice_list=compute_breakpoint(goal.width, 2, goal))

eval:
    self.breakpoint = self.break_choice[0]
    self.pin_set = self.break_choice[1]
    self.second_pin_set = goal.pin[self.breakpoint:]
    self.second_breakpoint = goal.width - self.breakpoint

    if self.breakpoint <= 2 or self.second_breakpoint <= 2:
        planner.fail("Width too narrow to divide further!")

subgoals:
    mux = Mux(width=2, db=goal.db, cost_vs_delay=goal.cost_vs_delay,
              Benmark=goal.Benmark)

subgoals:
    csa1 = Adder(width=self.breakpoint, pin=self.pin_set, carryin=goal.carryin,
                 db=goal.db, cost_vs_delay=goal.cost_vs_delay,
                 Benmark=goal.Benmark)

subgoals:
    csa2 = Adder(width=self.second_breakpoint, pin=self.second_pin_set, carryin=0,
                 db=goal.db, cost_vs_delay=goal.cost_vs_delay,
                 Benmark=goal.Benmark)


subgoals:
    csa3 = Adder(width=self.second_breakpoint, pin=self.second_pin_set, carryin=1,
                 db=goal.db, cost_vs_delay=goal.cost_vs_delay,
                 Benmark=goal.Benmark)


eval:
    solution.cost = adding_cost(planner, subgoals.mux, subgoals.csa1,
                                subgoals.csa2, subgoals.csa3)
    solution.delay = max(subgoals.csa1.delay, subgoals.csa2.delay,
                         subgoals.csa3.delay) + subgoals.mux.delay


commit:
    if subgoals.csa2.is_virtual or subgoals.csa3.is_virtual:
        solution.component = {}
    elif not subgoals.csa2.is_shared and not subgoals.csa3.is_shared:
        if subgoals.csa2.component.name.find("CSA") == -1 and \
               subgoals.csa1.component.name.find("CSA") == -1:
            sub_comp = {"sub_adder": subgoals.csa1.component,
                    "mux2": subgoals.mux.component}
            fa = adderRCA_CSAlo("adderlo"+str(goal.width),
                                component=sub_comp,
                                parameters={"width":goal.width,
                                            "divide":self.breakpoint-1})
            fa.initialize()
            solution.component = fa.instantiate()
        else:
            if subgoals.csa2.component == {}:
                hi_comp = subgoals.csa3.component
            else:
                hi_comp = subgoals.csa2.component
            sub_comp = {"csa_lo": subgoals.csa1.component,
                        "csa_hi": hi_comp,
                        "mux2": subgoals.mux.component}
            fa = adder_CSAlo("adderlo"+str(goal.width),
                                component=sub_comp,
                                parameters={"width":goal.width,
                                            "divide":self.breakpoint-1})
            fa.initialize()
            solution.component = fa.instantiate()
    elif subgoals.csa2.is_shared or subgoals.csa3.is_shared:
        if subgoals.csa2.component.name.find("CSA") == -1 and \
               subgoals.csa1.component.name.find("CSA") == -1:

            sub_comp = {"sub_adder": subgoals.csa1.component,
                        "mux2": subgoals.mux.component}
            fa = adderRCA_CSAhi("adderhi"+str(goal.width),
                                component=sub_comp,
                                parameters={"width":goal.width,
                                            "divide":self.breakpoint-1})
            fa.initialize()
            solution.component = fa.instantiate()
        elif subgoals.csa1.component.name.find("hi") != -1:

            hi_comp = subgoals.csa1.component
            sub_comp = {"csa_hi": hi_comp,
                        "mux2": subgoals.mux.component}
            fa = adder_CSAhihi("adderhi"+str(goal.width),
                                component=sub_comp,
                                parameters={"width":goal.width,
                                            "divide":self.breakpoint-1})
            fa.initialize()
            solution.component = fa.instantiate()
        else:
            if subgoals.csa2.component == {}:
                hi_comp = subgoals.csa3.component
                else:
                    hi_comp = subgoals.csa2.component
                sub_comp = {"csa_lo": subgoals.csa1.component,
                            "csa_hi": hi_comp,
                            "mux2": subgoals.mux.component}
                fa = adder_CSAhi("adderhi"+str(goal.width),
                                    component=sub_comp,
                                    parameters={"width":goal.width,
                                                "divide":self.breakpoint-1})
                fa.initialize()
                solution.component = fa.instantiate()
```

Add.goal

```
name:Add
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Add_type1.teq

```
from jsim_test import adder
to Add(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0
        self.pin = []
        for i in range(goal.width):
            self.pin.append(i)

    subgoals:
        adder = Adder(width=goal.width, pin=self.pin, carryin=0, db=goal.db,
                      cost_vs_delay=goal.cost_vs_delay,
                      Benmark=goal.Benmark)

    subgoals:
        v_output = v_output(width=goal.width, db=goal.db,
                            cost_vs_delay=goal.cost_vs_delay,
                            Benmark=goal.Benmark)
        z_output = z_logic(width=goal.width, db=goal.db,
                           cost_vs_delay=goal.cost_vs_delay,
                            Benmark=goal.Benmark)
        xor_array = subgoal_array(goal=gate_width, width=2, number=goal.width,
                                  func="xor", db=goal.db,
                                  cost_vs_delay=goal.cost_vs_delay,
                                  Benmark=goal.Benmark)
    eval:
        solution.cost = subgoals.adder.cost + subgoals.v_output.cost + \
                        subgoals.z_output.cost + subgoals.xor_array[0].cost *\
                        len(subgoals.xor_array)
        solution.delay = subgoals.adder.delay

    commit:
        sub_comp = {"sub_adder": subgoals.adder.component,
                    "z_circuit": subgoals.z_output.component,
                    "v_circuit": subgoals.v_output.component,
                    "xor2": subgoals.xor_array[0].component}
        fa = adder("adder32", component=sub_comp,
                   parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

ALU.goal

```
name:ALU
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

ALU_type1.teq

```
from jsim_test import ALU
to ALU(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        add = Add(width=goal.width, db=goal.db,
                  cost_vs_delay=goal.cost_vs_delay,
                  Benmark=goal.Benmark)
    subgoals:
        boole = boole(width=goal.width, db=goal.db,
                      cost_vs_delay=goal.cost_vs_delay,
                      Benmark=goal.Benmark)
    subgoals:
        shift = shifter(width=goal.width, db=goal.db,
                        cost_vs_delay=goal.cost_vs_delay,
                        Benmark=goal.Benmark)
    subgoals:
        compare = Compare(width=goal.width, db=goal.db,
                          cost_vs_delay=goal.cost_vs_delay,
                          Benmark=goal.Benmark)
    subgoals:
        mux_array = subgoal_array(goal=Mux, number=goal.width, width=4,
                                  db=goal.db, cost_vs_delay=goal.cost_vs_delay,
                                  Benmark=goal.Benmark)
    eval:
        solution.cost = subgoals.add.cost + subgoals.boole.cost + \
                        subgoals.shift.cost + subgoals.compare.cost + \
                        subgoals.mux_array[0].cost *goal.width
        solution.delay = max(subgoals.add.delay, subgoals.shift.delay) + \
                         subgoals.mux_array[0].delay

    commit:
        sub_comp = {"mux4": subgoals.mux_array[0].component,
                    "adder": subgoals.add.component,
                    "boole": subgoals.boole.component,
                    "shift": subgoals.shift.component,
                    "compare": subgoals.compare.component}
        fa = ALU("alu", component=sub_comp, parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

beta.goal

```
name:beta
properties:delay cost
functional attributes: width
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

beta_pipelined.teq

```
from jsim_test import beta_pipe

to beta(width, cost_vs_delay, Benmark, db):
    via pipeline:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        pc = PC(width=goal.width, db=goal.db, Benmark=goal.Benmark,
                cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        ctl = ctl(width=goal.width, db=goal.db, Benmark=goal.Benmark,
                  cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        regfile = reg(width=goal.width, db=goal.db, Benmark=goal.Benmark,
                      cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        alu = ALU(width=goal.width, db=goal.db, Benmark=goal.Benmark,
                  cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        mux2_array = subgoal_array(goal=Mux, number=goal.width * 2 + 5,
                                   width=2, db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)
        mux4_array = subgoal_array(goal=Mux, number=goal.width,
                                   width=4, db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)

        dreg_array = subgoal_array(goal=gate_fix_width, number=goal.width,
                                   func="dreg", db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)
        buffer_array = subgoal_array(goal=gate_fix_width, number=goal.width,
                                     func="buffer_2", db=goal.db,
                                     Benmark=goal.Benmark,
                                     cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        pcmux5 = pcmux(width=goal.width, db=goal.db,
                       Benmark=goal.Benmark,
                   cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        pcselz = pcselz(width=goal.width, db=goal.db,
                        Benmark=goal.Benmark,
                        cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        z_output = z_logic(width=goal.width, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.delay += subgoals.alu.delay + \
                          max(subgoals.pcmux5.delay, subgoals.ctl.delay +
                              subgoals.regfile.delay) + \
                              subgoals.mux4_array[0].delay
        solution.cost += subgoals.pc.cost + subgoals.ctl.cost + \
                         subgoals.regfile.cost + subgoals.alu.cost +\
                         subgoals.mux2_array[0].cost * (goal.width*2+5) \
                         + subgoals.mux4_array[0].cost * goal.width + \
                         subgoals.pcselz.cost + subgoals.pcmux5.cost\
                         + subgoals.z_output.cost + \
                         subgoals.dreg_array[0].cost *goal.width + \
                         subgoals.buffer_array[0].cost * goal.width

    commit:
        sub_comp = {"pc": subgoals.pc.component,
                    "ctl": subgoals.ctl.component,
                    "regfile": subgoals.regfile.component,
                    "alu": subgoals.alu.component,
                    "mux2": subgoals.mux2_array[0].component,
                    "mux4": subgoals.mux4_array[0].component,
                    "pcmux5": subgoals.pcmux5.component,
                    "pcselz": subgoals.pcselz.component,
                    "z_output": subgoals.z_output.component,
                    "dreg": subgoals.dreg_array[0].component,
                    "buffer_2": subgoals.buffer_array[0].component}
        fa = beta_pipe("beta", component=sub_comp, parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

beta_unpipelined.teq

```
from jsim_test import beta_unpipe
to beta(width, cost_vs_delay, Benmark, db):
```

```
via unpiplined:

first:
    solution.delay = 0
    solution.cost = 0

subgoals:
    pc = PC(width=goal.width, db=goal.db, Benmark=goal.Benmark,
            cost_vs_delay=goal.cost_vs_delay)

subgoals:
    ctl = ctl(width=goal.width, db=goal.db, Benmark=goal.Benmark,
            cost_vs_delay=goal.cost_vs_delay)

subgoals:
    regfile = reg(width=goal.width, db=goal.db, Benmark=goal.Benmark,
            cost_vs_delay=goal.cost_vs_delay)

subgoals:
    alu = ALU(width=goal.width, db=goal.db, Benmark=goal.Benmark,
            cost_vs_delay=goal.cost_vs_delay)

subgoals:
    mux2_array = subgoal_array(goal=Mux, number=goal.width * 2 + 5,
                               width=2, db=goal.db, Benmark=goal.Benmark,
                               cost_vs_delay=goal.cost_vs_delay)
    mux4_array = subgoal_array(goal=Mux, number=goal.width, width=4,
                               db=goal.db, Benmark=goal.Benmark,
                               cost_vs_delay=goal.cost_vs_delay)

subgoals:
    pcmux5 = pcmux(width=goal.width, db=goal.db,
                   Benmark=goal.Benmark,
                   cost_vs_delay=goal.cost_vs_delay)

subgoals:
    pcselz = pcselz(width=goal.width, db=goal.db,
                    Benmark=goal.Benmark,
                    cost_vs_delay=goal.cost_vs_delay)

subgoals:
    z_output = z_logic(width=goal.width, db=goal.db,
                       Benmark=goal.Benmark,
                       cost_vs_delay=goal.cost_vs_delay)

eval:
    solution.delay += subgoals.pc.delay + subgoals.alu.delay + \
                      max(subgoals.pcmux5.delay, subgoals.ctl.delay +
                          subgoals.regfile.delay) + \
                          subgoals.mux4_array[0].delay
        solution.cost += subgoals.pc.cost + subgoals.ctl.cost + \
                         subgoals.regfile.cost + subgoals.alu.cost +\
                         subgoals.mux2_array[0].cost * (goal.width*2+5) + \
                         subgoals.mux4_array[0].cost + subgoals.pcselz.cost + \
                         subgoals.pcmux5.cost + subgoals.z_output.cost

    commit:
        sub_comp = {"pc": subgoals.pc.component,
                    "ctl": subgoals.ctl.component,
                    "regfile": subgoals.regfile.component,
                    "alu": subgoals.alu.component,
                    "mux2": subgoals.mux2_array[0].component,
                    "mux4": subgoals.mux4_array[0].component,
                    "pcmux5": subgoals.pcmux5.component,
                    "pcselz": subgoals.pcselz.component,
                    "z_output": subgoals.z_output.component}

        fa = beta_unpipe("beta", component=sub_comp, parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

boole.goal

```
name:boole
properties:delay cost
functional attributes: width
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

boole_type1.teq

```
from jsim_test import boole
to boole(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        mux_array = subgoal_array(goal=Mux, number=goal.width,
                                    width=4, db=goal.db,
                                    Benmark=goal.Benmark,
                                    cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.mux_array[0].cost * goal.width
        solution.delay = subgoals.mux_array[0].delay
    commit:
        sub_comp = {"mux4": subgoals.mux_array[0].component}
        fa = boole("boole32", component=sub_comp,
                   parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

Compare.goal

```
name:Compare
properties:delay cost
functional attributes: width
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Compare_type1.teq

```
from jsim_test import compare
to Compare(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        xor2 = gate_width(func="xor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)
        nor2 = gate_width(func="nor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)
        not_gate = gate_fix_width(func="not", db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
        mux = Mux(width=4, db=goal.db,
                  Benmark=goal.Benmark,
                cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.xor2.cost + subgoals.nor2.cost + \
                        subgoals.not_gate.cost + subgoals.mux.cost
        solution.delay = subgoals.xor2.delay + subgoals.nor2.delay +\
                         subgoals.not_gate.delay + subgoals.mux.delay
    commit:
        sub_comp = {"xor2": subgoals.xor2.component,
                    "nor2": subgoals.nor2.component,
                    "inverter": subgoals.not_gate.component,
                    "mux4": subgoals.mux.component}
        fa = compare("compare32", component=sub_comp,
                     parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

ctl.goal

```
name:ctl
properties:delay cost
functional attributes: width
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

ctl_mix.teq

```
from jsim_test import ctl_mix
from jsim_component import cal_reg
to ctl(width, db, Benmark, cost_vs_delay):
    via mix:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        nor3_array = subgoal_array(goal=gate_width, number=3,
                                   func="nor", width=3, db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)
    subgoals:
        nand2 = gate_width(func="nand", width=2, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)
```

```
subgoals:
    nand3 = gate_width(func="nand", width=3, db=goal.db,
                       Benmark=goal.Benmark,
                       cost_vs_delay=goal.cost_vs_delay)
subgoals:
    nand4_array = subgoal_array(goal=gate_width, number=3,
                                func="nand", width=4, db=goal.db,
                                Benmark=goal.Benmark,
                                cost_vs_delay=goal.cost_vs_delay)
subgoals:
    nor2_array = subgoal_array(goal=gate_width, number=4,
                               func="nor", width=2, db=goal.db,
                               Benmark=goal.Benmark,
                               cost_vs_delay=goal.cost_vs_delay)
subgoals:
    nor4 = gate_width(func="nor", width=4, db=goal.db,
                      Benmark=goal.Benmark,
                      cost_vs_delay=goal.cost_vs_delay)
subgoals:
    not_array = subgoal_array(goal=gate_fix_width, func="not",
                              number=6, db=goal.db,
                              Benmark=goal.Benmark,
                              cost_vs_delay=goal.cost_vs_delay)
subgoals:
    xnor = gate_fix_width(func="xnor", db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)
subgoals:
    xor_array = subgoal_array(goal=gate_width, number=2,
                              func="xor", width=2, db=goal.db,
                              Benmark=goal.Benmark,
                              cost_vs_delay=goal.cost_vs_delay)

eval:
    solution.delay += subgoals.nor3_array[0].delay + subgoals.nand3.delay + \
                      subgoals.not_array[0].delay * 2 + subgoals.nor2_array[0].delay
    solution.cost += subgoals.nor3_array[0].cost * 3 + subgoals.nand2.cost + \
                     subgoals.nand3.cost + subgoals.nand4_array[0].cost * 3 + \
                     subgoals.nor2_array[0].cost * 4 + subgoals.nor4.cost + \
                     subgoals.not_array[0].cost * 6 + subgoals.xnor.cost + \
                     subgoals.xor_array[0].cost * 6

    c, d = cal_reg(ports=1, read=1, write=0, width=8, addr=6,
                   nlocations=64)
    solution.delay += d
    solution.cost += c
    commit:
        sub_comp = {"nor3": subgoals.nor3_array[0].component,
                    "nand3": subgoals.nand3.component,
                    "inverter": subgoals.not_array[0].component,
                    "nor2": subgoals.nor2_array[0].component,
                    "nand2": subgoals.nand2.component,
                    "xor2": subgoals.xor_array[0].component,
                    "nor4": subgoals.nor4.component,
                    "nand4": subgoals.nand4_array[0].component,
                    "xnor2": subgoals.xnor.component}
        fa = ctl_mix("ctl", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

ctl_rom.teq

```
from jsim_test import ctl_rom
from jsim_component import cal_reg

to ctl(width, db, Benmark, cost_vs_delay):
    via rom:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        nor2 = gate_width(func="nor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        not_gate = gate_fix_width(func="not", db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.delay += subgoals.nor2.delay + subgoals.not_gate.delay
        solution.cost += subgoals.nor2.cost + subgoals.not_gate.cost

        c, d = cal_reg(ports=1, read=1, write=0, width=18, addr=6,
                       nlocations=64)
        solution.delay += d
        solution.cost += c

    commit:
        sub_comp = {"inverter": subgoals.not_gate.component,
                "nor2": subgoals.nor2.component}
        fa = ctl_rom("ctl", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

FullAdder.goal

```
name:FullAdder
properties:delay cost c_delay s_delay
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

FullAdder_type1.teq

```
from jsim_test import fulladder
to FullAdder(db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.c_delay = 0
        solution.s_delay = 0
        solution.cost = 0
        solution.delay = 0
    subgoals:
        xor_array = subgoal_array(goal=gate_width, number=2, func="xor",
                                  width=2, db=goal.db, Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)

        nand2_array = subgoal_array(goal=gate_width, number=3, func="nand",
                                    width=2, db=goal.db,
                                    Benmark=goal.Benmark,
                                    cost_vs_delay=goal.cost_vs_delay)

        nand3 = gate_width(func="nand", width=3, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.c_delay = subgoals.nand2_array[0].delay + subgoals.nand3.delay
        solution.s_delay = subgoals.xor_array[0].delay * 2
        solution.cost = subgoals.xor_array[0].cost * 2 + \
                        subgoals.nand2_array[0].cost* 3 + subgoals.nand3.cost
        solution.delay = max(solution.c_delay, solution.s_delay)
    commit:
        sub_comp = {"xor2": subgoals.xor_array[0].component,
                    "nand2": subgoals.nand2_array[0].component,
                    "nand3": subgoals.nand3.component}
        fa = fulladder("fulladder", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

gate_width.goal

```
name:gate_width
properties:delay cost
functional attributes: func="" width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

gate_width_jsim.teq

```
from jsim_component import Connector, Pebble
to gate_width(func, width, Benmark, cost_vs_delay, db):
    via jsim:
    first:

        solution.cost = goal.db.query_cost(goal.func, width=goal.width)
        solution.delay = goal.db.query_delay(goal.func, width=goal.width)

    commit:
        in_con = []
        for i in goal.width:
            in_con.append(Connector(name="in" + str(i)))

        out = Connector(name="out")
        solution.component = Pebble(goal.func,
                                    input=in_con,
                                    output=[out])
```

gate_fix_width.goal

```
name:gate_fix_width
properties:delay cost
functional attributes: func
```

```
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

gate_fix_width_jsim.teq

```
from jsim_component import Connector, Pebble
to gate_fix_width(func, db, Benmark, cost_vs_delay):
    via jsim:
    first:
        solution.cost = goal.db.query_cost(func=goal.func)
        solution.delay = goal.db.query_delay(func=goal.func)

    commit:
        in_con = []
        width = goal.db.query_width(func=goal.func)
        for i in width:
            in_con.append(Connector(name="in" + str(i)))

        out = Connector(name="out")
        solution.component = Pebble(goal.func,
                                    input=in_con,
                                    output=[out])
```

HA.goal

```
name:HA
properties:delay cost
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

HA_adder.teq

```
from jsim_test import ha
to HA(db, Benmark, cost_vs_delay):
    via adder:

    first:
        solution.delay = 0
        solution.cost = 0
    subgoals:
        not_gate = gate_fix_width(func="not", db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        xor2 = gate_width(func="xor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        nand2 = gate_width(func="nand", width=2, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.delay += subgoals.xor2.delay
        solution.cost += subgoals.xor2.cost + subgoals.not_gate.cost + \
                         subgoals.nand2.cost

    commit:
        sub_comp = {"xor2": subgoals.xor2.component,
                    "nand2": subgoals.nand2.component,
                    "inverter": subgoals.not_gate.component}

        fa = ha("ha", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

Incrementer.goal

```
name:Incrementer
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Incrementer_adder.teq

```
from jsim_test import adder_incrementer
to Incrementer(width, db, Benmark, cost_vs_delay):
    via adder:
```

```
    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        not_gate = gate_fix_width(func="not", db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        ha = subgoal_array(goal=HA, number=goal.width-3, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.delay += subgoals.ha[0].delay * (goal.width-3)
        solution.cost += subgoals.ha[0].cost * (goal.width-3) + \
                         subgoals.not_gate.cost

    commit:
        sub_comp = {"ha": subgoals.ha[0].component,
                    "inverter": subgoals.not_gate.component}
        fa = adder_incrementer("adder_incrementer",
                               component=sub_comp,
                               parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

Incrementer_type1.teq

```
from jsim_test import incrementer
to Incrementer(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
       nand_array = subgoal_array(goal=gate_width, func="nand",
                                  number=goal.width-3, width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
    subgoals:
        not_array = subgoal_array(goal=gate_fix_width, func="not",
                                  number=goal.width-3, db=goal.db,
                                 Benmark=goal.Benmark,
                                 cost_vs_delay=goal.cost_vs_delay)
    subgoals:
        xor_array = subgoal_array(goal=gate_width, func="xor",
                                 number=goal.width-2, width=2,
                                 db=goal.db,
                                 Benmark=goal.Benmark,
                                 cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.delay += subgoals.nand_array[0].delay * (goal.width-3) + \
                          subgoals.not_array[0].delay * (goal.width-3) + \
                          subgoals.xor_array[0].delay
        solution.cost += subgoals.nand_array[0].cost * (goal.width-3) + \
                         subgoals.not_array[0].cost * (goal.width-3) + \
                         subgoals.xor_array[0].cost * (goal.width-2)

    commit:
        sub_comp = {"nand2": subgoals.nand_array[0].component,
                    "inverter": subgoals.not_array[0].component,
                    "xor2": subgoals.xor_array[0].component}

        fa = incrementer("incrementer", component=sub_comp,
                         parameters={"width":goal.width})
        fa.initialize()

        solution.component = fa.instantiate()
```

leftshifter.goal

```
name: leftshifter
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

leftshifter_type1.teq

```
from jsim_test import leftshifter
import math
to leftshifter(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0
        self.level = int(math.log(goal.width, 2))

    subgoals:
        mux_array = subgoal_array(goal=Mux, number=self.level * goal.width,
                                  width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
    eval:
        solution.cost = subgoals.mux_array[0].cost * self.level * goal.width
        solution.delay = subgoals.mux_array[0].delay * self.level

    commit:
        sub_comp = {"mux2": subgoals.mux_array[0].component}
        fa = leftshifter("leftshifter", component=sub_comp,
                         parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

memory.goal

```
name: memory
properties:delay cost
functional attributes: width=0  nlocations=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

memory_one_port.teq

```
from jsim_test import memory_one
from jsim_component import cal_reg

to memory(width, nlocations, db, Benmark, cost_vs_delay):
    via one_port:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        mem_split = mem_port_mux(width=goal.width, db=goal.db,
                                 Benmark=goal.Benmark,
                                 cost_vs_delay=goal.cost_vs_delay)


    eval:
        c, d = cal_reg(ports=2, read=2, write=1, width=goal.width,
                       addr=10, nlocations=goal.nlocations)
        solution.delay += d + subgoals.mem_split.delay
        solution.cost += c + subgoals.mem_split.cost

    commit:
solution.circuit = memory_one(mem_split=subgoals.mem_split.component)
```

<u>memory_two_port.teq</u>

```
from jsim_test import memory_two
from jsim_component import cal_reg

to memory(width, nlocations, db, Benmark, cost_vs_delay):
    via two_port:

    first:
        solution.delay = 0
        solution.cost = 0

    eval:
        c, d = cal_reg(ports=3, read=2, write=1, width=goal.width,
                       addr=10, nlocations=goal.nlocations)
        solution.delay += d
        solution.cost += c

    commit:
solution.circuit = memory_two()
```

<u>mem_port_mux.goal</u>

```
name: mem_port_mux
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

mem_port_mux_type1.teq

```
from jsim_test import mem_port_split
to mem_port_mux(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        tristate_array = subgoal_array(goal=gate_fix_width,
                                       func="tristate", number=goal.width*2,
                                       db=goal.db, Benmark=goal.Benmark,
                                       cost_vs_delay=goal.cost_vs_delay)
    subgoals:
        mux_array = subgoal_array(goal=Mux, number=3, width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
    subgoals:
        not_array = subgoal_array(goal=gate_fix_width, number=2, func="not",
                                  db=goal.db, Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
    subgoals:
        nor2_array = subgoal_array(goal=gate_width, number=2, func="nor",
                                   width=2, db=goal.db, Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.cost = subgoals.mux_array[0].cost * 3 + \
                        subgoals.not_array[0].cost * 2 + \
                        subgoals.nor2_array[0].cost * 2 + \
                        subgoals.tristate_array[0].cost * goal.width * 2
        solution.delay = subgoals.mux_array[0].delay + \
                         subgoals.not_array[0].delay + \
                         subgoals.tristate_array[0].delay

    commit:
        sub_comp = {"mux2": subgoals.mux_array[0].component,
                    "inverter": subgoals.not_array[0].component,
                    "nor2": subgoals.nor2_array[0].component,
                    "tristate": subgoals.tristate_array[0].component}
        fa = mem_port_split("mem_port", component=sub_comp,
                            parameters={"width":32})
        fa.initialize()
        solution.component = fa.instantiate()
```

Mux5.goal

```
name: Mux5
properties:delay cost
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None \
nlocations=0
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Mux5_type1.teq

```
from jsim_test import mux5
to Mux5(db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        mux4 = Mux(width=4, db=goal.db, Benmark=goal.Benmark,
                   cost_vs_delay=goal.cost_vs_delay)

        mux2 = Mux(width=2, db=goal.db, Benmark=goal.Benmark,
                   cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.delay += subgoals.mux2.delay + subgoals.mux4.delay
        solution.cost += subgoals.mux2.cost + subgoals.mux4.cost

    commit:
        sub_comp = {"mux2": subgoals.mux2.component,
                    "mux4": subgoals.mux4.component}
        fa = mux5("mux5", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

Mux.goal

```
name:Mux
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

Mux_jsim.teq

```
from jsim_component import Connector, Pebble
to Mux(width, Benmark, cost_vs_delay, db):
    via jsim:
    first:

        solution.cost = goal.db.query_cost("mux", width=goal.width)
        solution.delay = goal.db.query_delay("mux", width=goal.width)

    commit:
        if goal.width == 2:
            w = 3
        else:
            w = 6
        in_con = []
        for i in w:
            in_con.append(Connector(name="in" + str(i)))

        out = Connector(name="out")
        solution.component = Pebble(goal.func,
                                    input=in_con,
                                    output=[out])
```

NegFullAdder.goal

```
name:NegFullAdder
properties:delay cost c_delay s_delay
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

NegFullAdder_type1.teq

```
from jsim_test import negFA
to NegFullAdder(, Benmark, cost_vs_delay, db):
    via type1:

    first:
        solution.c_delay = 0
        solution.s_delay = 0
        solution.cost = 0
        solution.delay = 0

    subgoals:
        fa = FullAdder(db=goal.db,
                       Benmark=goal.Benmark,
                       cost_vs_delay=goal.cost_vs_delay)

        not_gate = gate_fix_width(func="not", db=goal.db,
                              Benmark=goal.Benmark,
                              cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.c_delay = subgoals.fa.c_delay + subgoals.not_gate.delay
        solution.s_delay = subgoals.fa.s_delay
        solution.cost = subgoals.fa.cost + subgoals.not_gate.cost
        solution.delay = max(solution.c_delay, solution.s_delay)

    commit:
        sub_comp = {"fulladder": subgoals.fa.component,
                    "inverter": subgoals.not_gate.component}
        fa = negFA("negfulladder", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

NegHalfAdder.goal

```
name:NegHalfAdder
properties:delay cost
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

NegHalfAdder_type1.teq

```
from jsim_test import neghalfadder
to NegHalfAdder(db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        nor2 = gate_width(func="nor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)
        nand2 = gate_width(func="nand", width=2, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.nor2.cost + subgoals.nand2.cost
        solution.delay = subgoals.nor2.delay

    commit:
sub_comp = {"nor2": subgoals.nor2.component, "nand2": subgoals.nand2.component}
fa = neghalfadder("neghalfadder", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

NegPosCarryOp.goal

```
name:NegPosCarryOp
properties:delay cost
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

NegPosCarryOp_type1.teq

```
from jsim_test import negposcarryop
to NegPosCarryOp(db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        nor2 = gate_width(func="nor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)

        oai = gate_fix_width(func="oai", db=goal.db,
                             Benmark=goal.Benmark,
                             cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.cost = subgoals.oai.cost + subgoals.nor2.cost
        solution.delay = max(subgoals.nor2.delay, subgoals.oai.delay)

    commit:
        sub_comp = {"nor2": subgoals.nor2.component, "oai21": subgoals.oai.component}
        fa = negposcarryop("negposcarryop", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

pcmux.goal

```
name:pcmux
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

pcmux_type1.teq

```
from jsim_test import pcmux5
to pcmux(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.delay = 0
        solution.cost = 0
        self.pin = []
        for i in range(goal.width):
            self.pin.append(i)

    subgoals:
        not_gate = gate_fix_width(func="not", db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
        nand2 = gate_width(func="nand", width=2, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)

        mux5 = Mux5(db=goal.db,
                    Benmark=goal.Benmark,
                    cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        adder = Adder(width=goal.width, pin=self.pin, carryin=0, db=goal.db,
                      Benmark=goal.Benmark,
                      cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.delay += subgoals.adder.delay + subgoals.mux5.delay
        solution.cost += subgoals.adder.cost + subgoals.mux5.cost +\
                         subgoals.not_gate.cost + subgoals.nand2.cost

    commit:
        sub_comp = {"adder": subgoals.adder.component,
                    "mux5": subgoals.mux5.component,
                    "inverter": subgoals.not_gate.component,
                    "nand2": subgoals.nand2.component}
        fa = pcmux5("pcmux5", component=sub_comp,
                    parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

pcselz.goal

```
name:pcselz
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

pcselz_type1.teq

```
from jsim_test import pcselz
to pcselz(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        not_gate = gate_fix_width(func="not", db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
        nor2 = gate_width(func="nor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)
        mux4 = Mux(width=4, db=goal.db,
                   Benmark=goal.Benmark,
                   cost_vs_delay=goal.cost_vs_delay)
        xor2 = gate_width(func="xor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.delay += subgoals.xor2.delay + subgoals.mux4.delay
        solution.cost += subgoals.not_gate.cost + subgoals.mux4.cost + \
                         subgoals.xor2.cost + subgoals.nor2.cost

    commit:
        sub_comp = {"inverter": subgoals.not_gate.component,
                    "nor2": subgoals.nor2.component,
                    "xor2": subgoals.xor2.component,
                    "mux4": subgoals.mux4.component}
        fa = pcselz("pcselz", component=sub_comp, parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

PC.goal

```
name:PC
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

PC_type1.teq

```
from jsim_test import pc
import functools
to PC(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        dreg_array = subgoal_array(goal=gate_fix_width, func="dreg",
                                   number=goal.width-2, db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)

        mux_array = subgoal_array(goal=Mux, number=goal.width, width=2,
                                  db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)

        increm = Incrementer(width=goal.width, db=goal.db,
                             Benmark=goal.Benmark,
                             cost_vs_delay=goal.cost_vs_delay)


    subgoals:
        instruct_mem = memory(width=goal.width, nlocations=1024, db=goal.db,
                              Benmark=goal.Benmark,
                              cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.delay += subgoals.instruct_mem.delay + subgoals.dreg_array[0].delay

        solution.cost += subgoals.dreg_array[0].cost * (goal.width-2) + \
                         subgoals.mux_array[0].cost * goal.width + \
                         subgoals.increm.cost
        if not subgoals.instruct_mem.is_virtual:
            solution.cost += subgoals.instruct_mem.cost

    commit:
        sub_comp = {"incrementer": subgoals.increm.component,
                    "dreg": subgoals.dreg_array[0].component,
                    "mux2": subgoals.mux_array[0].component}

        fa = pc("pc", component=sub_comp, parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

PosNegCarryOp.goal

```
name:PosNegCarryOp
```

```
properties:delay cost
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

PosNegCarryOp_type1.teq

```
from jsim_test import posnegcarryop
to PosNegCarryOp(db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        nand2 = gate_width(func="nand", width=2, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)

        aoi = gate_fix_width(func="aoi", db=goal.db,
                             Benmark=goal.Benmark,
                             cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.cost = subgoals.aoi.cost + subgoals.nand2.cost
        solution.delay = max(subgoals.nand2.delay, subgoals.aoi.delay)

    commit:
        sub_comp = {"nand2": subgoals.nand2.component, "aoi21": subgoals.aoi.component}
        fa = posnegcarryop("posnegcarryop", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

processor.goal

```
name: processor
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

processor_beta.teq

```
from jsim_component import output_circuit, Connector
import functools
to processor(width, db, Benmark, cost_vs_delay):
    via beta:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        mem = memory(width=goal.width, nlocations=1024, db=goal.db,
                     Benmark=goal.Benmark,
                     cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        beta = beta(width=goal.width, db=goal.db,
                    Benmark=goal.Benmark,
                    cost_vs_delay=goal.cost_vs_delay)

    eval:

        solution.delay += subgoals.mem.delay + subgoals.beta.delay
        solution.cost += subgoals.mem.cost + subgoals.beta.cost

    commit:
        beta = subgoals.beta.component
        beta_circuit = beta_processor(beta)

        cir = ""
        clk = Connector(name="clk")
        reset = Connector(name="reset")
        qid = Connector(name="id", width=32)
        mrd = Connector(name="mrd", width=32)
        ia = Connector(name="ia", width=32)
        ma = Connector(name="ma", width=32)
        moe = Connector(name="moe")
        wr = Connector(name="wr")
        werf = Connector(name="werf")
        mwd = Connector(name="mwd", width=32)

        mem_circuit = subgoals.mem.circuit

cir += beta_circuit.create(clk=clk,
                    reset=reset,
                    qid=qid,
                    mrd=mrd,
                  ia=ia,
                  ma=ma,
                  moe=moe,
                  wr=wr,
                  werf=werf,
                  mwd=mwd)
        cir += mem_circuit.create(clk=clk,
                  reset=reset,
                  qid=qid,
                  mrd=mrd,
                  ia=ia,
                  ma=ma,
                  moe=moe,
                  wr=wr,
                  werf=werf,
                  mwd=mwd)
        solution.component = output_circuit(cir)
```

reg.goal

```
name: reg
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

reg_type1.teq

```
from jsim_test import regfile
from jsim_component import cal_reg

to reg(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.delay = 0
        solution.cost = 0

    subgoals:
        mux_array = subgoal_array(goal=Mux, number=goal.width+5,
                                  width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
        nand2_array = subgoal_array(goal=gate_width, func="nand", number=2,
                                    width=2, db=goal.db,
                                    Benmark=goal.Benmark,
                                    cost_vs_delay=goal.cost_vs_delay)

        nand3_array = subgoal_array(goal=gate_width, func="nand", number=2,
                                    width=3, db=goal.db,
                                    Benmark=goal.Benmark,
                                    cost_vs_delay=goal.cost_vs_delay)

        nor_array = subgoal_array(goal=gate_width, func="nor", number=2,
                                  width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.delay += subgoals.mux_array[0].delay * 2
        solution.cost += subgoals.mux_array[0].cost * (goal.width+5) + \
                         subgoals.nand2_array[0].cost * (2) + \
                         subgoals.nand3_array[0].cost * (2) + \
                         subgoals.nor_array[0].cost * (2)

        c, d = cal_reg(ports=3, read=2, write=1, width=32,
                       addr=5, nlocations=31)
        solution.delay += d
        solution.cost += c

    commit:
        sub_comp = {"mux2": subgoals.mux_array[0].component,
                    "nand2": subgoals.nand2_array[0].component,
                    "nand3": subgoals.nand3_array[0].component,
                    "nor2": subgoals.nor_array[0].component}
        fa = regfile("regfile", component=sub_comp,
                     parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

rightshifter.goal

```
name: rightshifter
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
```

```
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

rightshifter_type1.teq

```
from jsim_test import rightshifter
import math
to rightshifter(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0
        self.level = int(math.log(goal.width, 2))

    subgoals:
        mux_array = subgoal_array(goal=Mux, number=self.level * goal.width + 1,
                                  width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.mux_array[0].cost * (self.level * goal.width + 1)
        solution.delay = subgoals.mux_array[0].delay * self.level

    commit:
        sub_comp = {"mux2": subgoals.mux_array[0].component}
        fa = rightshifter("rightshifter", component=sub_comp,
                          parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

shifter.goal

```
name: shifter
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

shifter_one.teq

```
from jsim_test import smallshifter
to shifter(width, db, Benmark, cost_vs_delay):
    via one:
```

```
    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        ls = leftshifter(width=goal.width, db=goal.db,
                         Benmark=goal.Benmark,
                         cost_vs_delay=goal.cost_vs_delay)
    subgoals:
        mux_array = subgoal_array(goal=Mux, number=goal.width * 2 + 1,
                                  width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.ls.cost + subgoals.mux_array[0].cost \
                      *(goal.width*2 + 1)
        solution.delay = subgoals.ls.delay + subgoals.mux_array[0].delay * 2

    commit:
        sub_comp = {"mux2": subgoals.mux_array[0].component,
                    "leftshifter":subgoals.ls.component}
        fa = smallshifter("shifter32", component=sub_comp,
                          parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

shifter_two.teq

```
from jsim_test import shifter
to shifter(width, db, Benmark, cost_vs_delay):
    via two:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        ls = leftshifter(width=goal.width, db=goal.db,
                         Benmark=goal.Benmark,
                         cost_vs_delay=goal.cost_vs_delay)

    subgoals:
        rs = rightshifter(width=goal.width, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)
        mux_array = subgoal_array(goal=Mux, number=goal.width,
                                  width=2, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.cost = subgoals.ls.cost + subgoals.rs.cost + \
                        subgoals.mux_array[0].cost *goal.width
        solution.delay = max(subgoals.ls.delay, subgoals.rs.delay) + \
                         subgoals.mux_array[0].delay

    commit:
        sub_comp = {"mux2": subgoals.mux_array[0].component,
                    "rightshifter": subgoals.rs.component,
                    "leftshifter":subgoals.ls.component}
        fa = shifter("shifter32", component=sub_comp,
                     parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

v_output.goal

```
name: v_output
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

v_output_type1.teq

```
from jsim_test import v_output
to v_output(width, db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0


    subgoals:
        inverter_array = subgoal_array(goal=gate_fix_width,
                                       func="not", number=3,
                                        db=goal.db,
                                        Benmark=goal.Benmark,
                                        cost_vs_delay=goal.cost_vs_delay)

        nand3_array = subgoal_array(goal=gate_width, func="nand",
                                    width=3, number=2, db=goal.db,
                                    Benmark=goal.Benmark,
                                    cost_vs_delay=goal.cost_vs_delay)
        nand2 = gate_width(func="nand", width=2, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.inverter_array[0].cost * len(subgoals.inverter_array) +
                        subgoals.nand3_array[0].cost * len(subgoals.nand3_array)+ \
                        subgoals.nand2.cost
        solution.delay = subgoals.inverter_array[0].delay + \
                         subgoals.nand3_array[0].delay + \
                         subgoals.nand2.delay

    commit:
        sub_comp = {"inverter": subgoals.inverter_array[0].component,
                    "nand3": subgoals.nand3_array[0].component,
                    "nand2": subgoals.nand2.component}
        fa = v_output("v_output", component=sub_comp)
        fa.initialize()
        solution.component = fa.instantiate()
```

XOR3.goal

```
name: XOR3
properties:delay cost
functional attributes:
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

XOR3_type1.teq

```
from jsim_test import xor3
to XOR3(db, Benmark, cost_vs_delay):
    via type1:

    first:
        solution.cost = 0
        solution.delay = 0

    subgoals:
        xor2 = subgoal_array(goal=gate_width, func="xor", number=2,
                             width=2, db=goal.db,
                             Benmark=goal.Benmark,
                             cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.xor2[0].cost * 2
        solution.delay = subgoals.xor2[0].delay * 2

    commit:
        sub_comp = {"xor2": subgoals.xor2[0].component}
        fa = xor3("xor3", component=sub_comp)
        fa.initialize()
        solution.component= fa.instantiate()
```

z_logic.goal

```
name: z_logic
properties:delay cost
functional attributes: width=0
non-functional attributes: cost_vs_delay=0 Benmark=False db=None
evaluation:
if Benmark:
    satisfaction = 1/(delay * cost)
else:
    satisfaction = 1/(cost * cost_vs_delay + delay * (1 - cost_vs_delay))
```

z_logic_type16.teq

```
from jsim_test import z_output
to z_logic(width, db, Benmark, cost_vs_delay):
    via type16:

    first:
        solution.cost = 0
        solution.delay = 0
        if goal.width == 16:
            self.nor4_no = 4
        else:
            planner.fail("this is not 16")

    subgoals:
        nor4_array = subgoal_array(goal=gate_width, func="nor",
                                  width=4, number=self.nor4_no, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
        nand2 = gate_width(func="nand", width=4, db=goal.db,
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)
        not_gate = gate_fix_width(func="not", db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.nor4_array[0].cost * len(subgoals.nor4_array) +\
                        subgoals.nand2.cost + subgoals.not_gate.cost
        solution.delay = subgoals.nor4_array[0].delay + subgoals.nand2.delay +\
                         subgoals.not_gate.delay

    commit:
        sub_comp = {"nor4": subgoals.nor4_array[0].component,
                    "nand4": subgoals.nand4_array[0].component,
                    "nor2": subgoals.nor2.component}
        fa = z_output("z_output", component=sub_comp,
                      parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

z_logic_type32.teq

```
from jsim_test import z_output
to z_logic(width, db, Benmark, cost_vs_delay):
    via type32:

    first:
        solution.cost = 0
        solution.delay = 0
        if goal.width == 32:
            self.nand4_no = goal.width/4/4
            self.nor4_no = goal.width/4
        else:
            planner.fail("this is not 32")

    subgoals:
        nor4_array = subgoal_array(goal=gate_width, func="nor", width=4,
                                   number=self.nor4_no, db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)
        nand4_array = subgoal_array(goal=gate_width, func="nand", width=4,
                                    number=self.nand4_no, db=goal.db,
                                  Benmark=goal.Benmark,
                                  cost_vs_delay=goal.cost_vs_delay)
        nor2 = gate_width(func="nor", width=2, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)

    eval:
        solution.cost = subgoals.nor4_array[0].cost * len(subgoals.nor4_array) + \
                        subgoals.nand4_array[0].cost * len(subgoals.nand4_array)+ \
                        subgoals.nor2.cost
        solution.delay = subgoals.nor4_array[0].delay + \
                         subgoals.nand4_array[0].delay + \
                         subgoals.nor2.delay

    commit:
        sub_comp = {"nor4": subgoals.nor4_array[0].component,
                    "nand4": subgoals.nand4_array[0].component,
                    "nor2": subgoals.nor2.component}
        fa = z_output("z_output", component=sub_comp, parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

<u>z_logic_type8.teq</u>

```
from jsim_test import z_output
to z_logic(width, db, Benmark, cost_vs_delay):
    via type8:

    first:
        solution.cost = 0
        solution.delay = 0
        if goal.width == 8:
            self.nor4_no = 2
        else:
            planner.fail("this is not 8")

    subgoals:
        nor4_array = subgoal_array(goal=gate_width, func="nor", width=4,
                                   number=self.nor4_no, db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)

        nand2 = gate_width(width=2, db=goal.db, func="nand",
                           Benmark=goal.Benmark,
                           cost_vs_delay=goal.cost_vs_delay)

        not_gate = gate_fix_width(db=goal.db, func="not",
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)


    eval:
        solution.cost = subgoals.nor4_array[0].cost * len(subgoals.nor4_array) + \
                        subgoals.nand2.cost + subgoals.not_gate.cost
        solution.delay = subgoals.nor4_array[0].delay + \
                         subgoals.nand2.delay + subgoals.not_gate.delay

    commit:
        sub_comp = {"nor4": subgoals.nor4_array[0].component,
                    "nand4": subgoals.nand4_array[0].component,
                    "nor2": subgoals.nor2.component}
        fa = z_output("z_output", component=sub_comp,
                      parameters={"width":goal.width})
        fa.initialize()
        solution.component = fa.instantiate()
```

z_logic_type64.teq

```
from jsim_test import z_output
to z_logic(width, db, Benmark, cost_vs_delay):
    via type64:

    first:
        solution.cost = 0
        solution.delay = 0
        if goal.width == 64:
            self.nand4_no = goal.width/4/4
            self.nor4_no = goal.width/4
        else:
            planner.fail("this is not 64")

    subgoals:
        nor4_array = subgoal_array(goal=gate_width, func="nor",
                                   width=4, number=self.nor4_no, db=goal.db,
                                   Benmark=goal.Benmark,
                                   cost_vs_delay=goal.cost_vs_delay)
        nand4_array = subgoal_array(goal=gate_width, func="nand",
                                    width=4, number=self.nand4_no, db=goal.db,
                                    Benmark=goal.Benmark,
                                    cost_vs_delay=goal.cost_vs_delay)
        nor2 = gate_width(func="nor", width=4, db=goal.db,
                          Benmark=goal.Benmark,
                          cost_vs_delay=goal.cost_vs_delay)
```

```
eval:
    solution.cost = subgoals.nor4_array[0].cost * len(subgoals.nor4_array) + \
                    subgoals.nand4_array[0].cost * len(subgoals.nand4_array)+ \
                    subgoals.nor2.cost
    solution.delay = subgoals.nor4_array[0].delay + \
                     subgoals.nand4_array[0].delay + \
                     subgoals.nor2.delay

commit:
    sub_comp = {"nor4": subgoals.nor4_array[0].component,
                "nand4": subgoals.nand4_array[0].component,
                "nor2": subgoals.nor2.component}
    fa = z_output("z_output", component=sub_comp,
                  parameters={"width":goal.width})
    fa.initialize()
    solution.component = fa.instantiate()
```

Appendix C

JSim Implementations Generated by Fide

Highest Benmark

```
.include "nominal.jsim"
.include "stdcell.jsim"
.include "projcheckoff.jsim"

.subckt knex a b
.connect a b
.ends
.subckt xor3 a b c z
Xxor30 a b t xor2
Xxor31 t c z xor2
.ends

.subckt negfulladder a b c0 c1 s
Xnegfulladder0 a b c0 nout s fulladder
Xnegfulladder1 nout c1 inverter
.ends

.subckt incrementer ia[31:0] increment[31:0]
Xincrementer0 ia[1:0] increment[1:0] knex
.connect c2 vdd
Xincrementer1 c[30:2] ia[30:2] nc[31:3] nand2
Xincrementer2 nc[31:3] c[31:3] inverter
Xincrementer3 c[31:2] ia[31:2] increment[31:2] xor2
.ends

.subckt posnegcarryop g2 p2 g1 p1 ngout npout
Xposnegcarryop0 p1 p2 npout nand2
Xposnegcarryop1 g1 p2 g2 ngout aoi21
.ends
```

```
.subckt alu ALUFN[5:0] A[31:0] B[31:0] alu[31:0] z v n
Xalu0 ALUFN0 A[31:0] B[31:0] S[31:0] z v n adder32
Xalu1 ALUFN[3:0] A[31:0] B[31:0] boole[31:0] boole32
Xalu2 ALUFN[1:0] A[31:0] B[4:0] shift[31:0] shifter32
Xalu3 ALUFN[2:1] z v n compare[31:0] compare32
Xalu4 ALUFN5#32 ALUFN4#32 S[31:0] shift[31:0] boole[31:0] compare[31:0] alu[31:0] mux4
.ends

.subckt z_output S[31:0] z
Xz_output0 S[31:0] z0[7:0] nor4
Xz_output1 z0[7:0] z1[1:0] nand4
Xz_output2 z1[1:0] z nor2
.ends

.subckt v_output A31 B31 S31 v
Xv_output0 A31 na inverter
Xv_output1 B31 nb inverter
Xv_output2 S31 ns inverter
Xv_output3 A31 B31 ns fir nand3
Xv_output4 na nb S31 sed nand3
Xv_output5 fir sed v nand2
.ends

.subckt pc clk reset ini[31:0] ia[31:0] nextia[31:0] increment[31:0]
.connect ia1 0
.connect ia0 0
Xpc0 muxout[31:2] clk#30 ia[31:2] dreg
Xpc1 muxout[31:0] nextia[31:0] knex
Xpc2 reset#32 ini[31:0] vdd 0#31 muxout[31:0] mux2
Xpc3 0 ia[31:0] 0#29 vdd 0#2 increment[31:0] cdummy sub_adder
.ends

.subckt boole32 ALUFN[3:0] A[31:0] B[31:0] boole[31:0]
Xboole320 A[31:0] B[31:0] ALUFN0#32 ALUFN1#32 ALUFN2#32 ALUFN3#32 boole[31:0] mux4
.ends

.subckt regfile clk werf ra2sel ra[4:0] rb[4:0] rc[4:0] wdata[31:0]
+ radata[31:0] rbdata[31:0]
Xregfile0 ra2sel#5 rb[4:0] rc[4:0] ra2mux[4:0] mux2
Xregfile1 ra[4:3] nan0 nand2
Xregfile2 ra[2:0] nan1 nand3
Xregfile3 nan0 nan1 nra31 nor2
Xregfile4 ra2mux[4:3] nbn0 nand2
Xregfile5 ra2mux[2:0] nbn1 nand3
Xregfile6 nbn0 nbn1 nrb31 nor2
Xregfile7
```

```
+ vdd 0 0 ra[4:0] adata[31:0]
+ vdd 0 0 ra2mux[4:0] bdata[31:0]
+ 0 clk werf rc[4:0] wdata[31:0]
+ $memory width=32 nlocations=31
Xregfile8 nra31#32 adata[31:0] 0#32 radata[31:0] mux2
Xregfile9 nrb31#32 bdata[31:0] 0#32 rbdata[31:0] mux2
.ends

.subckt fulladder a b c0 c1 s
Xfulladder0 a b g1 xor2
Xfulladder1 g1 c0 s xor2
Xfulladder2 a b g2 nand2
Xfulladder3 a c0 g3 nand2
Xfulladder4 b c0 g4 nand2
Xfulladder5 g2 g3 g4 c1 nand3
.ends

.subckt negposcarryop ng2 np2 ng1 np1 gout pout
Xnegposcarryop0 np1 np2 pout nor2
Xnegposcarryop1 ng1 np2 ng2 gout oai21
.ends

.subckt shifter32 ALUFN[1:0] A[31:0] B[4:0] shift[31:0]
Xshifter320 ALUFN1 0 A31 ctl mux2
Xshifter321 ALUFN0#32 A[31:0] A[0:31] ins[31:0] mux2
Xshifter322 ctl ins[31:0] B[4:0] outs[31:0] leftshifter
Xshifter323 ALUFN0#32 outs[31:0] outs[0:31] shift[31:0] mux2
.ends

.subckt adder32 ALUFN0 A[31:0] B[31:0] S[31:0] z v n
Xadder320 B[31:0] ALUFN0#32 bx[31:0] xor2
Xadder321 ALUFN0 A[31:0] bx[31:0] S[31:0] cdummy sub_adder
Xadder322 S[31:0] z z_output
.connect S31 n
Xadder323 A31 bx31 S31 v v_output
.ends

.subckt neghalfadder a b ng np
Xneghalfadder0 a b np nor2
Xneghalfadder1 a b ng nand2
.ends

.subckt pcselz xpcsel[2:0] id27 z irq pcsel[2:0]
Xpcselz0 xpcsel0 nxpcsel0 inverter
Xpcselz1 nxpcsel0 xpcsel1 btestout nor2
Xpcselz2 id27 z bzout xor2
Xpcselz3 irq#3 btestout#3 xpcsel[2:0] vdd 0#4 bzout vdd 0#2 pcsel[2:0] mux4
```

```
.ends

.subckt compare32 ALUFN[1:0] z v n cmp[31:0]
Xcompare320 n v gt xor2
Xcompare321 gt z nlet nor2
Xcompare322 nlet let inverter
Xcompare323 ALUFN[1:0] 0 gt z let cmp0 mux4
.connect 0 cmp[31:1]
.ends

.subckt beta clk reset irq nextid[31:0] mrd[31:0] nextia[31:0] ma[31:0]
+moe wr mwd[31:0] werf
Xbeta0 nextid[31:0] clk#32 id[31:0] dreg
Xbeta1 clk reset pcmux5out[31:0] ia[31:0] nextia[31:0] pcpf[31:0] pc
Xbeta2 reset id[31:26] ra2sel bsel alufn[5:0] wdsel[1:0] werf moe
+wr xpcsel[2:0] wasel asel ctl
Xbeta3 clk werf ra2sel ra[4:0] rb[4:0] rc[4:0] wdata[31:0] radata[31:0]
+rbdata[31:0] regfile
Xbeta4 alufn[5:0] A[31:0] B[31:0] alu[31:0] aluz v n alu
Xbeta5 bsel#32 rbdata[31:0] id15#16 id[15:0] BB[31:0] mux2
Xbeta6 BB[31:0] B[31:0] buffer_8
Xbeta7 wdsel1#32 wdsel0#32 ia31 pcpf[30:0] mrd[31:0] alu[31:0] 0#32
+ wdata[31:0] mux4
Xbeta8 pcsel[2:0] id[15:0] pcpf[31:0] radata[31:0] ia31
+ pcmux5out[31:0] beq[31:0] pcmux5
Xbeta9 xpcsel[2:0] id27 z irq pcsel[2:0] pcselz
Xbeta10 wasel#5 xrc[4:0] vdd#4 0 rc[4:0] mux2
Xbeta11 asel#32 radata[31:0] 0 beq[30:0] A[31:0] mux2
Xbeta12 radata[31:0] z z_output
Xbeta13 alu[31:0] ma[31:0] knex
Xbeta14 rbdata[31:0] mwd[31:0] knex
Xbeta15 id[15:11] rb[4:0] knex
Xbeta16 id[25:21] xrc[4:0] knex
Xbeta17 id[20:16] ra[4:0] knex
.ends

.subckt leftshifter ctl A[31:0] B[4:0] shift[31:0]
Xleftshifter0 B4#32 A[31:0] A[15:0] ctl#16 w[31:0] mux2
Xleftshifter1 B3#32 w[31:0] w[23:0] ctl#8 x[31:0] mux2
Xleftshifter2 B2#32 x[31:0] x[27:0] ctl#4 y[31:0] mux2
Xleftshifter3 B1#32 y[31:0] y[29:0] ctl#2 z[31:0] mux2
Xleftshifter4 B0#32 z[31:0] z[30:0] ctl shift[31:0] mux2
.ends

.subckt mux5 s0 s1 s2 d0 d1 d2 d3 d4 out
Xmux50 s1 s2 d0 d1 d2 d3 out1 mux4
Xmux51 s0 out1 d4 out mux2
```

```
.ends

.subckt ctl reset id31 id30 id29 id28 id27 id26 ra2sel bsel alufn[5:0]
+ wdsel[1:0] werf moe wr pcsel[2:0] wasel asel
Xctl0
+ vdd 0 0 id31 id30 id29 id28 id27 id26 alufn[5:0] bsel wdsel0
+ $memory width=8 nlocations=64 contents=(
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b00000010 //18 ld
+  0b00000010 //19 str
+  0b11100000
+  0b00000000 //1b jmp
+  0b11100000
+  0b00000000 //1d beq
+  0b00000000 //1e bne
+  0b01101000 //1f ldr
+  0b00000001 //20 add
+  0b00000101 //21 sub
+  0b00001001 //22 mul
+  0b00001101 //23 div
+  0b11001101 //24 cmpeq
+  0b11010101 //25 cmplt
+  0b11011101 //26 cmple
+  0b11100000 //
+  0b01100001 //28 and
+  0b01111001 //29 or
+  0b01011001 //2a xor
+  0b11100000 //
+  0b10000001 //2c shl
+  0b10000101 //2d shr
+  0b10001101 //2e sra
+  0b11100000
+  0b00000011 //30 add
+  0b00000111 //31 sub
+  0b00001011 //32 mul
+  0b01001111 //33 div
+  0b11001111 //34 cmpeq
+  0b11010111 //35 cmplt
+  0b11011111 //36 cmple
+  0b11100000
+  0b01100011 //38 and
+  0b01111011 //39 or
+  0b01011011 //3a xor
+  0b11100000
+  0b10000011 //3c shl
+  0b10000111 //3d shr
+  0b10001111 //3e sra
+  0b11100000)
Xctl1 id31 id28 id27 nisst0 nor3
Xctl2 id30 id29 id26 isst1 nand3
Xctl3 nisst0 isst0 inverter
Xctl4 isst1 isst0 isst nor2
Xctl5 isst werf inverter
Xctl6 isst xwr knex
Xctl7 ra2sel isst knex
Xctl8 id30 id29 isld1 nand2
Xctl9 id28 id27 isld2 xor2
Xctl10 id27 id26 isld3 xor2
Xctl11 id31 isld1 isld2 isld3 isld nor4
Xctl12 isld moe knex
Xctl13 xwr nxwr inverter
Xctl14 nxwr reset wr nor2
.connect 0 wasel
Xctl15 id29 id28 id27 id26 isldr0 nand4
Xctl16 id30 nid30 inverter
Xctl17 id31 nid30 isldr0 isldr nor3
Xctl18 isldr asel knex
Xctl19 isldr isld nwdsel1 nor2
Xctl20 nwdsel1 wdsel1 inverter
Xctl21 id30 id29 id27 id26 isj0 nand4
Xctl22 id31 id28 isj0 isjmp nor3
```

```
Xctl23 pcsel1 isjmp knex
Xctl24 id31 nid31 inverter
Xctl25 nid31 id30 id29 id28 isbe0 nand4
Xctl26 id27 id26 isbe1 xnor2
Xctl27 isbe0 isbe1 be nor2
Xctl28 pcsel0 be knex
.connect 0 pcsel2
.ends

.subckt pcmux5 pcsel[2:0] id[15:0] ia[31:0] radata[31:0] pc31 out[31:0]
+ beq[31:0]
Xpcmux50 radata31 pc31 npc31 nand2
Xpcmux51 npc31 jpc31 inverter
Xpcmux52 0 ia[31:0] ia31 id15#13 id[15:0] 0#2 beq[31:0] cdummy sub_adder
Xpcmux53 pcsel2#32 pcsel1#32 pcsel0#32 pc31 ia[30:0] jpc31 radata[30:2] 0#2 pc31 beq[30:0
.ends

.subckt sub_adder Cin A[31:0] B[31:0] S[31:0] Cout
.connect p0_0 vdd
Xsub_adder0 A[31:1] B[31:1] g0_[31:1] p0_[31:1] neghalfadder
Xsub_adder1 A0 B0 Cin g0_0 S0 negfulladder
Xsub_adder2 g0_[31:1] p0_[31:1] g0_[30:0] p0_[30:0] g1_[31:1] p1_[31:1] negposcarryop
Xsub_adder3 g0_[0:0] p0_[0:0] g1_[0:0] p1_[0:0] inverter
Xsub_adder4 g1_[31:2] p1_[31:2] g1_[29:0] p1_[29:0] g2_[31:2] p2_[31:2] posnegcarryop
Xsub_adder5 g1_[0:1] p1_[0:1] g2_[0:1] p2_[0:1] inverter
Xsub_adder6 g2_[31:4] p2_[31:4] g2_[27:0] p2_[27:0] g3_[31:4] p3_[31:4] negposcarryop
Xsub_adder7 g2_[0:3] p2_[0:3] g3_[0:3] p3_[0:3] inverter
Xsub_adder8 g3_[31:8] p3_[31:8] g3_[23:0] p3_[23:0] g4_[31:8] p4_[31:8] posnegcarryop
Xsub_adder9 g3_[0:7] p3_[0:7] g4_[0:7] p4_[0:7] inverter
Xsub_adder10 g4_[31:16] p4_[31:16] g4_[15:0] p4_[15:0] g5_[31:16] p5_[31:16] negposcarryo
Xsub_adder11 g4_[0:15] p4_[0:15] g5_[0:15] p5_[0:15] inverter
Xsub_adder12 A[31:1] B[31:1] g5_[30:0] S[31:1] xor3
.connect g5_31 Cout
.ends

Xbbb clk reset 0 id[31:0] mrd[31:0] ia[31:0] ma[31:0] moe wr mwd[31:0] werf beta
.subckt mem_port moe wr clk werf reset mwd[31:0] port[2:0] mrd[31:0] rw[31:0]
Xmem_port0 werf 0 moe port0 mux2
Xmem_port1 clk nclk inverter
Xmem_port2 werf nclk 0 port1 mux2
Xmem_port3 wr nwr inverter
Xmem_port4 reset nwr port2out nor2
Xmem_port5 werf port2out 0 port2 mux2
Xmem_port6 reset werf nwerf nor2
Xmem_port7 nwerf#32 mwd[31:0] rw[31:0] tristate
Xmem_port8 werf#32 rw[31:0] mrd[31:0] tristate
.ends
```

```
Xm_port moe wr clk werf reset mwd[31:0] mport[2:0] mrd[31:0] rw[31:0] mem_port
Xmem
+ vdd 0 0 ia[11:2] id[31:0]
+ mport0 mport1 mport2 ma[11:2] rw[31:0]
+ $memory width=32 nlocations=1024 file="/mit/6.004/jsim/projcheckoff.bin"
Vclk clk 0 pulse(3.3,0,4.585000ns,.01ns,.01ns,4.585000ns)
Vreset reset 0 pwl(0ns 3.3v, 9.190000ns 3.3v, 9.200000ns 0v)
.tran 8666ns
```

The fastest implementation

```
.include "nominal.jsim"
.include "stdcell.jsim"
.include "projcheckoff.jsim"

.subckt knex a b
.connect a b
.ends
.subckt xor3 a b c z
Xxor30 a b t xor2
Xxor31 t c z xor2
.ends

.subckt negfulladder a b c0 c1 s
Xnegfulladder0 a b c0 nout s fulladder
Xnegfulladder1 nout c1 inverter
.ends

.subckt incrementer ia[31:0] increment[31:0]
Xincrementer0 ia[1:0] increment[1:0] knex
.connect c2 vdd
Xincrementer1 c[30:2] ia[30:2] nc[31:3] nand2
Xincrementer2 nc[31:3] c[31:3] inverter
Xincrementer3 c[31:2] ia[31:2] increment[31:2] xor2
.ends

.subckt posnegcarryop g2 p2 g1 p1 ngout npout
Xposnegcarryop0 p1 p2 npout nand2
Xposnegcarryop1 g1 p2 g2 ngout aoi21
.ends

.subckt alu ALUFN[5:0] A[31:0] B[31:0] alu[31:0] z v n
Xalu0 ALUFN0 A[31:0] B[31:0] S[31:0] z v n adder32
Xalu1 ALUFN[3:0] A[31:0] B[31:0] boole[31:0] boole32
Xalu2 ALUFN[1:0] A[31:0] B[4:0] shift[31:0] shifter32
Xalu3 ALUFN[2:1] z v n compare[31:0] compare32
```

```
Xalu4 ALUFN5#32 ALUFN4#32 S[31:0] shift[31:0] boole[31:0] compare[31:0]
+ alu[31:0] mux4
.ends

.subckt z_output S[31:0] z
Xz_output0 S[31:0] z0[7:0] nor4
Xz_output1 z0[7:0] z1[1:0] nand4
Xz_output2 z1[1:0] z nor2
.ends

.subckt v_output A31 B31 S31 v
Xv_output0 A31 na inverter
Xv_output1 B31 nb inverter
Xv_output2 S31 ns inverter
Xv_output3 A31 B31 ns fir nand3
Xv_output4 na nb S31 sed nand3
Xv_output5 fir sed v nand2
.ends

.subckt pc clk reset ini[31:0] ia[31:0] nextia[31:0] increment[31:0]
.connect ia1 0
.connect ia0 0
Xpc0 muxout[31:2] clk#30 ia[31:2] dreg
Xpc1 muxout[31:0] nextia[31:0] knex
Xpc2 reset#32 ini[31:0] vdd 0#31 muxout[31:0] mux2
Xpc3 ia[31:0] increment[31:0] incrementer
.ends

.subckt boole32 ALUFN[3:0] A[31:0] B[31:0] boole[31:0]
Xboole320 A[31:0] B[31:0] ALUFN0#32 ALUFN1#32 ALUFN2#32 ALUFN3#32
+boole[31:0] mux4
.ends

.subckt regfile clk werf ra2sel ra[4:0] rb[4:0] rc[4:0] wdata[31:0]
+ radata[31:0] rbdata[31:0]
Xregfile0 ra2sel#5 rb[4:0] rc[4:0] ra2mux[4:0] mux2
Xregfile1 ra[4:3] nan0 nand2
Xregfile2 ra[2:0] nan1 nand3
Xregfile3 nan0 nan1 nra31 nor2
Xregfile4 ra2mux[4:3] nbn0 nand2
Xregfile5 ra2mux[2:0] nbn1 nand3
Xregfile6 nbn0 nbn1 nrb31 nor2
Xregfile7
+ vdd 0 0 ra[4:0] adata[31:0]
+ vdd 0 0 ra2mux[4:0] bdata[31:0]
+ 0 clk werf rc[4:0] wdata[31:0]
+ $memory width=32 nlocations=31
```

```
Xregfile8 nra31#32 adata[31:0] 0#32 radata[31:0] mux2
Xregfile9 nrb31#32 bdata[31:0] 0#32 rbdata[31:0] mux2
.ends

.subckt fulladder a b c0 c1 s
Xfulladder0 a b g1 xor2
Xfulladder1 g1 c0 s xor2
Xfulladder2 a b g2 nand2
Xfulladder3 a c0 g3 nand2
Xfulladder4 b c0 g4 nand2
Xfulladder5 g2 g3 g4 c1 nand3
.ends

.subckt negposcarryop ng2 np2 ng1 np1 gout pout
Xnegposcarryop0 np1 np2 pout nor2
Xnegposcarryop1 ng1 np2 ng2 gout oai21
.ends

.subckt shifter32 ALUFN[1:0] A[31:0] B[4:0] shift[31:0]
Xshifter320 0 A[31:0] B[4:0] l[31:0] leftshifter
Xshifter321 ALUFN1 A[31:0] B[4:0] r[31:0] rightshifter
Xshifter322 ALUFN0#32 l[31:0] r[31:0] shift[31:0] mux2
.ends

.subckt adder32 ALUFN0 A[31:0] B[31:0] S[31:0] z v n
Xadder320 B[31:0] ALUFN0#32 bx[31:0] xor2
Xadder321 ALUFN0 A[31:0] bx[31:0] S[31:0] cdummy sub_adder
Xadder322 S[31:0] z z_output
.connect S31 n
Xadder323 A31 bx31 S31 v v_output
.ends

.subckt neghalfadder a b ng np
Xneghalfadder0 a b np nor2
Xneghalfadder1 a b ng nand2
.ends

.subckt pcselz xpcsel[2:0] id27 z irq pcsel[2:0]
Xpcselz0 xpcsel0 nxpcsel0 inverter
Xpcselz1 nxpcsel0 xpcsel1 btestout nor2
Xpcselz2 id27 z bzout xor2
Xpcselz3 irq#3 btestout#3 xpcsel[2:0] vdd 0#4 bzout vdd 0#2
+ pcsel[2:0] mux4
.ends

.subckt compare32 ALUFN[1:0] z v n cmp[31:0]
Xcompare320 n v gt xor2
```

```
Xcompare321 gt z nlet nor2
Xcompare322 nlet let inverter
Xcompare323 ALUFN[1:0] 0 gt z let cmp0 mux4
.connect 0 cmp[31:1]
.ends

.subckt beta clk reset irq nextid[31:0] mrd[31:0] nextia[31:0]
+ ma[31:0] moe wr mwd[31:0] werf
Xbeta0 nextid[31:0] clk#32 id[31:0] dreg
Xbeta1 clk reset pcmux5out[31:0] ia[31:0] nextia[31:0] pcpf[31:0] pc
Xbeta2 reset id[31:26] ra2sel bsel alufn[5:0] wdsel[1:0] werf moe wr
+ xpcsel[2:0] wasel asel ctl
Xbeta3 clk werf ra2sel ra[4:0] rb[4:0] rc[4:0] wdata[31:0]
+radata[31:0] rbdata[31:0] regfile
Xbeta4 alufn[5:0] A[31:0] B[31:0] alu[31:0] aluz v n alu
Xbeta5 bsel#32 rbdata[31:0] id15#16 id[15:0] BB[31:0] mux2
Xbeta6 BB[31:0] B[31:0] buffer_8
Xbeta7 wdsel1#32 wdsel0#32 ia31 pcpf[30:0] mrd[31:0] alu[31:0] 0#32
+wdata[31:0] mux4
Xbeta8 pcsel[2:0] id[15:0] pcpf[31:0] radata[31:0] ia31 pcmux5out[31:0]
+beq[31:0] pcmux5
Xbeta9 xpcsel[2:0] id27 z irq pcsel[2:0] pcselz
Xbeta10 wasel#5 xrc[4:0] vdd#4 0 rc[4:0] mux2
Xbeta11 asel#32 radata[31:0] 0 beq[30:0] A[31:0] mux2
Xbeta12 radata[31:0] z z_output
Xbeta13 alu[31:0] ma[31:0] knex
Xbeta14 rbdata[31:0] mwd[31:0] knex
Xbeta15 id[15:11] rb[4:0] knex
Xbeta16 id[25:21] xrc[4:0] knex
Xbeta17 id[20:16] ra[4:0] knex
.ends

.subckt rightshifter ctl A[31:0] B[4:0] shift[31:0]
Xrightshifter0 ctl 0 A31 sign mux2
Xrightshifter1 B4#32 A[31:0] sign#16 A[31:16] w[31:0] mux2
Xrightshifter2 B3#32 w[31:0] sign#8 w[31:8] x[31:0] mux2
Xrightshifter3 B2#32 x[31:0] sign#4 x[31:4] y[31:0] mux2
Xrightshifter4 B1#32 y[31:0] sign#2 y[31:2] z[31:0] mux2
Xrightshifter5 B0#32 z[31:0] sign z[31:1] shift[31:0] mux2
.ends

.subckt leftshifter ctl A[31:0] B[4:0] shift[31:0]
Xleftshifter0 B4#32 A[31:0] A[15:0] ctl#16 w[31:0] mux2
Xleftshifter1 B3#32 w[31:0] w[23:0] ctl#8 x[31:0] mux2
Xleftshifter2 B2#32 x[31:0] x[27:0] ctl#4 y[31:0] mux2
Xleftshifter3 B1#32 y[31:0] y[29:0] ctl#2 z[31:0] mux2
Xleftshifter4 B0#32 z[31:0] z[30:0] ctl shift[31:0] mux2
```

```
.ends

.subckt mux5 s0 s1 s2 d0 d1 d2 d3 d4 out
Xmux50 s1 s2 d0 d1 d2 d3 out1 mux4
Xmux51 s0 out1 d4 out mux2
.ends

.subckt ctl reset id31 id30 id29 id28 id27 id26 ra2sel bsel alufn[5:0]
+wdsel[1:0] werf moe wr pcsel[2:0] wasel asel
Xctl0
+ vdd 0 0 id31 id30 id29 id28 id27 id26 alufn[5:0] werf bsel wdsel[1:0]
+xwr ra2sel pcsel[2:0] asel wasel moe
+ $memory width=18 nlocations=64 contents=(
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000000000000000
+  0b000000111000000001
+  0b000000010011000000
+  0b000000000000000000
+  0b000000100000010000
+  0b000000000000000000
+  0b000000100000001000
+  0b000000100000001000
+  0b011010101000000101
+  0b000000100100000000 //20 add
+  0b000001100100000000 //21 sub
+  0b000010100100000000 //22 mul
+  0b000011100100000000 //23 div
+  0b110011100100000000 //24 cmpeq
+  0b110101100100000000 //25 cmplt
+  0b110111100100000000 //26 cmple
+  0b000000000000000000 //
+  0b011000100100000000 //28 and
+  0b011110100100000000 //29 or
+  0b010110100100000000 //2a xor
+  0b000001000000000000 //2b
+  0b100000100100000000 //2c shl
+  0b100001100100000000 //2d shr
+  0b100011100100000000 //2e sra
+  0b000000000000000000
+  0b000000110100000000 //30 add
+  0b000001110100000000 //31 sub
+  0b000010110100000000 //32 mul
+  0b010011110100000000 //33 div
+  0b110011110100000000 //34 cmpeq
+  0b110101110100000000 //35 cmplt
+  0b110111110100000000 //36 cmple
+  0b000000000000000000
+  0b011000110100000000 //38 and
+  0b011110110100000000 //39 or
+  0b010110110100000000 //3a xor
+  0b000000000000000000
+  0b100000110100000000 //3c shl
+  0b100001110100000000 //3d shr
+  0b100011110100000000
+  0b000000000000000000)
Xctl1 xwr nxwr inverter
Xctl2 nxwr reset wr nor2
.ends

.subckt pcmux5 pcsel[2:0] id[15:0] ia[31:0] radata[31:0] pc31 out[31:0] beq[31:0]
Xpcmux50 radata31 pc31 npc31 nand2
Xpcmux51 npc31 jpc31 inverter
Xpcmux52 0 ia[31:0] ia31 id15#13 id[15:0] 0#2 beq[31:0] cdummy sub_adder
Xpcmux53 pcsel2#32 pcsel1#32 pcsel0#32 pc31 ia[30:0] jpc31 radata[30:2] 0#2 pc31 beq[30:0
.ends

.subckt sub_adder Cin A[31:0] B[31:0] S[31:0] Cout
.connect p0_0 vdd
Xsub_adder0 A[31:1] B[31:1] g0_[31:1] p0_[31:1] neghalfadder
Xsub_adder1 A0 B0 Cin g0_0 S0 negfulladder
Xsub_adder2 g0_[31:1] p0_[31:1] g0_[30:0] p0_[30:0] g1_[31:1] p1_[31:1] negposcarryop
Xsub_adder3 g0_[0:0] p0_[0:0] g1_[0:0] p1_[0:0] inverter
```

```
Xsub_adder4 g1_[31:2] p1_[31:2] g1_[29:0] p1_[29:0] g2_[31:2] p2_[31:2] posnegcarryop
Xsub_adder5 g1_[0:1] p1_[0:1] g2_[0:1] p2_[0:1] inverter
Xsub_adder6 g2_[31:4] p2_[31:4] g2_[27:0] p2_[27:0] g3_[31:4] p3_[31:4] negposcarryop
Xsub_adder7 g2_[0:3] p2_[0:3] g3_[0:3] p3_[0:3] inverter
Xsub_adder8 g3_[31:8] p3_[31:8] g3_[23:0] p3_[23:0] g4_[31:8] p4_[31:8] posnegcarryop
Xsub_adder9 g3_[0:7] p3_[0:7] g4_[0:7] p4_[0:7] inverter
Xsub_adder10 g4_[31:16] p4_[31:16] g4_[15:0] p4_[15:0] g5_[31:16] p5_[31:16] negposcarryo
Xsub_adder11 g4_[0:15] p4_[0:15] g5_[0:15] p5_[0:15] inverter
Xsub_adder12 A[31:1] B[31:1] g5_[30:0] S[31:1] xor3
.connect g5_31 Cout
.ends

Xbbb clk reset 0 id[31:0] mrd[31:0] ia[31:0] ma[31:0] moe wr mwd[31:0] +
werf beta
Xmem
+ vdd 0 0 ia[11:2] id[31:0]
+ moe 0 0 ma[11:2] mrd[31:0]
+ 0 clk wr ma[11:2] mwd[31:0]
+ $memory width=32 nlocations=1024 file="/mit/6.004/jsim/projcheckoff.bin"

Vclk clk 0 pulse(3.3,0,4.390000ns,.01ns,.01ns,4.390000ns)
Vreset reset 0 pwl(0ns 3.3v, 8.800000ns 3.3v, 8.810000ns 0v)
.tran 8298ns
```

The smallest implementation

```
.include "nominal.jsim"
.include "stdcell.jsim"
.include "projcheckoff.jsim"

.subckt knex a b
.connect a b
.ends
.subckt z_output S[31:0] z
Xz_output0 S[31:0] z0[7:0] nor4
Xz_output1 z0[7:0] z1[1:0] nand4
Xz_output2 z1[1:0] z nor2
.ends

.subckt alu ALUFN[5:0] A[31:0] B[31:0] alu[31:0] z v n
Xalu0 ALUFN0 A[31:0] B[31:0] S[31:0] z v n adder32
Xalu1 ALUFN[3:0] A[31:0] B[31:0] boole[31:0] boole32
Xalu2 ALUFN[1:0] A[31:0] B[4:0] shift[31:0] shifter32
Xalu3 ALUFN[2:1] z v n compare[31:0] compare32
Xalu4 ALUFN5#32 ALUFN4#32 S[31:0] shift[31:0] boole[31:0] compare[31:0]
+ alu[31:0] mux4
.ends
```

```
.subckt rca C0 A[31:0] B[31:0] S[31:0] C32
Xrca0 A0 B0 C0 c1 S0 fulladder
Xrca1 A[31:1] B[31:1] c[31:1] c[32:2] S[31:1] fulladder
.ends

.subckt v_output A31 B31 S31 v
Xv_output0 A31 na inverter
Xv_output1 B31 nb inverter
Xv_output2 S31 ns inverter
Xv_output3 A31 B31 ns fir nand3
Xv_output4 na nb S31 sed nand3
Xv_output5 fir sed v nand2
.ends

.subckt boole32 ALUFN[3:0] A[31:0] B[31:0] boole[31:0]
Xboole320 A[31:0] B[31:0] ALUFN0#32 ALUFN1#32 ALUFN2#32 ALUFN3#32 boole[31:0] mux4
.ends

.subckt adder_incrementer ia[31:0] increment[31:0]
Xadder_incrementer0 ia[1:0] increment[1:0] knex
Xadder_incrementer1 ia2 increment2 inverter
Xadder_incrementer2 ia3 ia2 increment3 c3 ha
Xadder_incrementer3 ia[31:4] c[30:3] increment[31:4] c[31:4] ha
.ends

.subckt pcselz xpcsel[2:0] id27 z irq pcsel[2:0]
Xpcselz0 xpcsel0 nxpcsel0 inverter
Xpcselz1 nxpcsel0 xpcsel1 btestout nor2
Xpcselz2 id27 z bzout xor2
Xpcselz3 irq#3 btestout#3 xpcsel[2:0] vdd 0#4 bzout vdd 0#2 pcsel[2:0] mux4
.ends

.subckt compare32 ALUFN[1:0] z v n cmp[31:0]
Xcompare320 n v gt xor2
Xcompare321 gt z nlet nor2
Xcompare322 nlet let inverter
Xcompare323 ALUFN[1:0] 0 gt z let cmp0 mux4
.connect 0 cmp[31:1]
.ends

.subckt pc clk reset ini[31:0] ia[31:0] nextia[31:0] increment[31:0]
.connect ia1 0
.connect ia0 0
Xpc0 muxout[31:2] clk#30 ia[31:2] dreg
Xpc1 muxout[31:0] nextia[31:0] knex
Xpc2 reset#32 ini[31:0] vdd 0#31 muxout[31:0] mux2
Xpc3 ia[31:0] increment[31:0] adder_incrementer
```

```
.ends

.subckt beta clk reset irq id[31:0] mrd[31:0] ia[31:0] ma[31:0] moe wr
+ mwd[31:0] werf
Xbeta0 clk reset pcmux5out[31:0] ia[31:0] nextia[31:0] pcpf[31:0] pc
Xbeta1 reset id[31:26] ra2sel bsel alufn[5:0] wdsel[1:0] werf moe wr
+xpcsel[2:0] wasel asel ctl
Xbeta2 clk werf ra2sel ra[4:0] rb[4:0] rc[4:0] wdata[31:0] radata[31:0]
+ rbdata[31:0] regfile
Xbeta3 alufn[5:0] A[31:0] B[31:0] alu[31:0] aluz v n alu
Xbeta4 bsel#32 rbdata[31:0] id15#16 id[15:0] B[31:0] mux2
Xbeta5 wdsel1#32 wdsel0#32 ia31 pcpf[30:0] mrd[31:0] alu[31:0] 0#32
+wdata[31:0] mux4
Xbeta6 pcsel[2:0] id[15:0] pcpf[31:0] radata[31:0] ia31 pcmux5out[31:0]
+beq[31:0] pcmux5
Xbeta7 xpcsel[2:0] id27 z irq pcsel[2:0] pcselz
Xbeta8 wasel#5 xrc[4:0] vdd#4 0 rc[4:0] mux2
Xbeta9 asel#32 radata[31:0] 0 beq[30:0] A[31:0] mux2
Xbeta10 radata[31:0] z z_output
Xbeta11 alu[31:0] ma[31:0] knex
Xbeta12 rbdata[31:0] mwd[31:0] knex
Xbeta13 id[15:11] rb[4:0] knex
Xbeta14 id[25:21] xrc[4:0] knex
Xbeta15 id[20:16] ra[4:0] knex
.ends

.subckt regfile clk werf ra2sel ra[4:0] rb[4:0] rc[4:0] wdata[31:0]
+radata[31:0] rbdata[31:0]
Xregfile0 ra2sel#5 rb[4:0] rc[4:0] ra2mux[4:0] mux2
Xregfile1 ra[4:3] nan0 nand2
Xregfile2 ra[2:0] nan1 nand3
Xregfile3 nan0 nan1 nra31 nor2
Xregfile4 ra2mux[4:3] nbn0 nand2
Xregfile5 ra2mux[2:0] nbn1 nand3
Xregfile6 nbn0 nbn1 nrb31 nor2
Xregfile7
+ vdd 0 0 ra[4:0] adata[31:0]
+ vdd 0 0 ra2mux[4:0] bdata[31:0]
+ 0 clk werf rc[4:0] wdata[31:0]
+ $memory width=32 nlocations=31
Xregfile8 nra31#32 adata[31:0] 0#32 radata[31:0] mux2
Xregfile9 nrb31#32 bdata[31:0] 0#32 rbdata[31:0] mux2
.ends

.subckt shifter32 ALUFN[1:0] A[31:0] B[4:0] shift[31:0]
Xshifter320 ALUFN1 0 A31 ctl mux2
Xshifter321 ALUFN0#32 A[31:0] A[0:31] ins[31:0] mux2
```

```
Xshifter322 ctl ins[31:0] B[4:0] outs[31:0] leftshifter
Xshifter323 ALUFN0#32 outs[31:0] outs[0:31] shift[31:0] mux2
.ends

.subckt ctl reset id31 id30 id29 id28 id27 id26 ra2sel bsel alufn[5:0] wdsel[1:0] werf mo
Xctl0
+ vdd 0 0 id31 id30 id29 id28 id27 id26 alufn[5:0] bsel wdsel0
+ $memory width=8 nlocations=64 contents=(
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b11100000
+  0b00000010 //18 ld
+  0b00000010 //19 str
+  0b11100000
+  0b00000000 //1b jmp
+  0b11100000
+  0b00000000 //1d beq
+  0b00000000 //1e bne
+  0b01101000 //1f ldr
+  0b00000001 //20 add
+  0b00000101 //21 sub
+  0b00001001 //22 mul
+  0b00001101 //23 div
+  0b11001101 //24 cmpeq
+  0b11010101 //25 cmplt
+  0b11011101 //26 cmple
+  0b11100000 //
+  0b01100001 //28 and
+  0b01111001 //29 or
+  0b01011001 //2a xor
+  0b11100000 //
+  0b10000001 //2c shl
+  0b10000101 //2d shr
+  0b10001101 //2e sra
+  0b11100000
+  0b00000011 //30 add
+  0b00000111 //31 sub
+  0b00001011 //32 mul
+  0b01001111 //33 div
+  0b11001111 //34 cmpeq
+  0b11010111 //35 cmplt
+  0b11011111 //36 cmple
+  0b11100000
+  0b01100011 //38 and
+  0b01111011 //39 or
+  0b01011011 //3a xor
+  0b11100000
+  0b10000011 //3c shl
+  0b10000111 //3d shr
+  0b10001111 //3e sra
+  0b11100000)
Xctl1 id31 id28 id27 nisst0 nor3
Xctl2 id30 id29 id26 isst1 nand3
Xctl3 nisst0 isst0 inverter
Xctl4 isst1 isst0 isst nor2
Xctl5 isst werf inverter
Xctl6 isst xwr knex
Xctl7 ra2sel isst knex
Xctl8 id30 id29 isld1 nand2
Xctl9 id28 id27 isld2 xor2
Xctl10 id27 id26 isld3 xor2
Xctl11 id31 isld1 isld2 isld3 isld nor4
Xctl12 isld moe knex
Xctl13 xwr nxwr inverter
Xctl14 nxwr reset wr nor2
.connect 0 wasel
Xctl15 id29 id28 id27 id26 isldr0 nand4
Xctl16 id30 nid30 inverter
Xctl17 id31 nid30 isldr0 isldr nor3
Xctl18 isldr asel knex
Xctl19 isldr isld nwdsel1 nor2
Xctl20 nwdsel1 wdsel1 inverter
Xctl21 id30 id29 id27 id26 isj0 nand4
```

```
Xctl22 id31 id28 isj0 isjmp nor3
Xctl23 pcsel1 isjmp knex
Xctl24 id31 nid31 inverter
Xctl25 nid31 id30 id29 id28 isbe0 nand4
Xctl26 id27 id26 isbe1 xnor2
Xctl27 isbe0 isbe1 be nor2
Xctl28 pcsel0 be knex
.connect 0 pcsel2
.ends

.subckt pcmux5 pcsel[2:0] id[15:0] ia[31:0] radata[31:0] pc31 out[31:0] beq[31:0]
Xpcmux50 radata31 pc31 npc31 nand2
Xpcmux51 npc31 jpc31 inverter
Xpcmux52 0 ia[31:0] ia31 id15#13 id[15:0] 0#2 beq[31:0] cdummy rca
Xpcmux53 pcsel2#32 pcsel1#32 pcsel0#32 pc31 ia[30:0] jpc31 radata[30:2] 0#2 pc31 beq[30:0
.ends

.subckt ha a c0 s c1
Xha0 a c0 s xor2
Xha1 a c0 nc1 nand2
Xha2 nc1 c1 inverter
.ends

.subckt adder32 ALUFN0 A[31:0] B[31:0] S[31:0] z v n
Xadder320 B[31:0] ALUFN0#32 bx[31:0] xor2
Xadder321 ALUFN0 A[31:0] bx[31:0] S[31:0] cdummy rca
Xadder322 S[31:0] z z_output
.connect S31 n
Xadder323 A31 bx31 S31 v v_output
.ends

.subckt leftshifter ctl A[31:0] B[4:0] shift[31:0]
Xleftshifter0 B4#32 A[31:0] A[15:0] ctl#16 w[31:0] mux2
Xleftshifter1 B3#32 w[31:0] w[23:0] ctl#8 x[31:0] mux2
Xleftshifter2 B2#32 x[31:0] x[27:0] ctl#4 y[31:0] mux2
Xleftshifter3 B1#32 y[31:0] y[29:0] ctl#2 z[31:0] mux2
Xleftshifter4 B0#32 z[31:0] z[30:0] ctl shift[31:0] mux2
.ends

.subckt fulladder a b c0 c1 s
Xfulladder0 a b g1 xor2
Xfulladder1 g1 c0 s xor2
Xfulladder2 a b g2 nand2
Xfulladder3 a c0 g3 nand2
Xfulladder4 b c0 g4 nand2
Xfulladder5 g2 g3 g4 c1 nand3
.ends
```

```
.subckt mux5 s0 s1 s2 d0 d1 d2 d3 d4 out
Xmux50 s1 s2 d0 d1 d2 d3 out1 mux4
Xmux51 s0 out1 d4 out mux2
.ends

Xbbb clk reset 0 id[31:0] mrd[31:0] ia[31:0] ma[31:0] moe wr mwd[31:0]
+werf beta
.subckt mem_port moe wr clk werf reset mwd[31:0] port[2:0] mrd[31:0] rw[31:0]
Xmem_port0 werf 0 moe port0 mux2
Xmem_port1 clk nclk inverter
Xmem_port2 werf nclk 0 port1 mux2
Xmem_port3 wr nwr inverter
Xmem_port4 reset nwr port2out nor2
Xmem_port5 werf port2out 0 port2 mux2
Xmem_port6 reset werf nwerf nor2
Xmem_port7 nwerf#32 mwd[31:0] rw[31:0] tristate
Xmem_port8 werf#32 rw[31:0] mrd[31:0] tristate
.ends

Xm_port moe wr clk werf reset mwd[31:0] mport[2:0] mrd[31:0] rw[31:0] mem_port
Xmem
+ vdd 0 0 ia[11:2] id[31:0]
+ mport0 mport1 mport2 ma[11:2] rw[31:0]
+ $memory width=32 nlocations=1024 file="/mit/6.004/jsim/projcheckoff.bin"

Vclk clk 0 pulse(3.3,0,8.740000ns,.01ns,.01ns,8.740000ns)
Vreset reset 0 pwl(0ns 3.3v, 17.500000ns 3.3v, 17.510000ns 0v)
.tran 16502ns
```

Bibliography

[1] H.S. Stone and P.M. Kogge. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Trans. Computers*, 22(8):786–793, Aug 1973.

[2] J. Mazzola Paluska. Automatic implementation generation for pervasive applications. *Master's thesis. Massachusetts Institute of Technology*, 2004.

[3] U. Saif, H. Pham, J. Mazzola Paluska, J. Waterman, C. Terman, and S. Ward. A case for goal-oriented programming semantics. In *System Support for Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing*, 2003.

[4] J. Mazzola Paluska, H. Pham, U. Saif, G. Chau, C. Terman, and S. Ward. Structured decomposition of adaptive applications. In *Proceedings of the 6th IEEE International Conference on Pervasive Computing and Communication*, March 2008.

[5] The JSim documentation. http://6004.csail.mit.edu/currentsemester/handouts/jsim.pdf.

[6] 6.004 website. http://6004.csail.mit.edu/.

[7] J. Mazzola Paluska, H. Pham, U. Saif, G. Chau, C. Terman, and S. Ward. Structured decomposition of adaptive applications. *Pervasive and Mobile Computing*, 2008.

[8] I. Page. Closing the gap between hardware and software: hardware-software cosynthesis at Oxford. *Hardware-Software Cosynthesis for Reconfigurable Systems (Digest No: 1996/036), IEE Colloquium on*, pages 2/1–211, 22 Feb 1996.

[9] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. *VLSI '99. Proceedings IEEE Computer Society Workshop On*, pages 68–73, 1999.

[10] R.K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *Design and Test of Computers, IEEE*, 10(3):29–41, Sep 1993.

[11] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 136–144, 19-21 Apr 1995.

[12] MyHDL - Python hardware description language. http://myhdl.jandecaluwe.com/doku.php.

[13] W. Luk and S. McKeever. Pebble: A language for parametrised and reconfigurable hardware design. In *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, pages 9–18, London, UK, 1998. Springer-Verlag.

[14] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification invited talk. In *MEMOCODE '03: Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, page 249, Washington, DC, USA, 2003. IEEE Computer Society.

[15] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[16] Deepinder P. Sidhu. Logic programming applied to hardware design specification and verification. *SIGMICRO Newsl.*, 15(4):309–313, 1984.

[17] S. P. Boyd and S. J. Kim. Geometric programming for circuit optimization. In *ISPD '05: Proceedings of the 2005 international symposium on Physical design*, pages 44–46, New York, NY, USA, 2005. ACM.

[18] S. S. Lin and Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 811–816, 4-8 June 2007.

[19] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):193–207, Mar 2005.

[20] D. Pham, E. Behnen, M. Bolliger, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, B. Le, Y. Masubuchi, S. Posluszny, M. Riley, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. The design methodology and implementation of a first-generation CELL processor: a multi-core SoC. *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pages 45–49, 18-21 Sept. 2005.

[21] A.K. Verma and P. Ienne. Towards the automatic exploration of arithmetic-circuit architectures. *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 445–450, 2006.

[22] D. Jackson. *Software Abstractions.* The MIT Press, 2006.