Design and Implementation of the Hitachi SuperH Processor Core

by

Bei Wang

B.S., Electrical Engineering (1996)
University of Nevada, Reno

Submitted to the Department of Electrical Engineering

and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science at the

Massachusetts Institute of Technology

May 1998

© 1998 Massachusetts Institute of Technology
All rights reserved.

Signature of Author

Department of Electrical Engineering and Computer Science
May 21, 1998

Certified by

Chris Terman
Senior Lecturer of EECS
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
ABSTRACT

The Hitachi SuperH RISC Engine (SH3) is a pipelined RISC microcontroller with a compact instruction set. At 32-bit data and 16-bit instruction lengths, this architecture must get around obstacles such as a small register file (16 general purpose registers) and short bit fields for specifying registers and data addresses. This thesis analyzes the SH3 architecture characteristics and describes an effort to build a datapath in VHDL for the SH3 to pipeline instructions using techniques such as forwarding and delayed branching.

Thesis Supervisor: Chris Terman
Title: Senior Lecturer of EECS Department
Acknowledgements

Many thanks to all those who helped me on this thesis: Dr. Chris Terman, Scott Adams, Chris Hanson, and the entire Amorphous group.

To all my friends here, Mike, Janet, Gloria, Coach, Ralph, Becca, I could not have done it without you. Thanks for making my MIT days so wonderful!

To my parents, I love you very much and thank you for being there for me every step of the way.
# Table of Contents

## Chapter 1: Introduction

1.1 Objective: ............................................. 5  
1.2 Background on Instruction Set Architecture ............................... 5  
1.3 Background on Pipelines: ........................................ 9  
1.4 Characteristics of RISC: ........................................ 14  
1.5 SH3 Instruction Set Architecture Analysis ................................ 14  
  1.5.1 Summary: .............................................. 14  
  1.5.2 Analysis: ................................................ 15  
  1.5.3 Exception Handling: ....................................... 19  
1.6 SPARC V8 Background ............................................. 21

## Chapter 2: Design and Implementation

2.1 SPARC datapath ............................................. 22  
2.2 Hitachi Datapath .............................................. 25  
  2.2.1 Datapath Summary: ....................................... 25  
  2.2.2 Placement of Registers: .................................... 29  
  2.2.3 Hitachi Design ........................................... 35  
  2.2.4 Tools: ................................................... 53

## Chapter 3: Future Work

...................................................... 54
Chapter 1: Introduction
1.1 Objective:

The AI Amorphous Computing Group needs a custom designed system-on-a-chip to demonstrate the idea of an intelligent material. The key to these computing elements is that they must be small but not necessarily very reliable. The prototype is to be a system on a die of area 25 mm² with a core processor, some 50-100K bytes of memory, and radio communication circuitry.

A decision was made to use the Hitachi SuperH RISC engine (SH3) as core processor. This processor is chosen because of its compact instruction set allows efficient use of program memory compared to many other RISC processors such as the SPARC. This thesis describes the design and implementation of the core logic which is mostly about datapath but with information useful for control logic design as well.

The design is to be done in VHDL, a hardware description language. The datapath will be simulated and the control logic will be simulated as well as synthesized.

1.2 Background on Instruction Set Architecture

According to the kind of internal storage they use, microprocessors can be classified roughly as stack, accumulator, and register architecture. Register architecture can be further divided into register-memory, register-register(load-store), and memory-memory architectures. Almost all
microprocessor architectures today are register architecture and most RISC architectures are load-store.

Load-store architecture rose in popularity because of register are faster than memory and are easier for compiler to use. When registers are used to hold variables, memory traffic is reduced, and code density improves since registers can be named with fewer bits than memory locations. The number of registers to implement are determined by the way compiler uses them. Most compilers reserve some registers for expression evaluation, parameter passing, and to hold variables. Typically load-store uses more registers than other forms of register architecture.

RISC machines now have at least 16 general purpose registers. Memory-memory machines have the advantage of compact number of instructions but often have large variations in code size. Register-register machines are at the other end of the spectrum with higher instruction count but simple, fixed-length instruction encoding. The rest of the discussion will focus on register-register type of architecture.

Memory addressing modes are the methods an architecture access data. They may be classified as the following:

<table>
<thead>
<tr>
<th>Modes</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Value is in a register</td>
</tr>
<tr>
<td>Immediate</td>
<td>const</td>
</tr>
<tr>
<td>Displacement</td>
<td>access a local variable</td>
</tr>
<tr>
<td>Register indirect</td>
<td>access using a pointer (addr of variable being pointed to)</td>
</tr>
</tbody>
</table>
Immediate, displacement, and register indirect addressing modes dominate addressing mode usage. These are similar but separate from branch offsets which are used to address instructions.

Major operations on the instruction set include the following categories:

- Arithmetic and logical
- Data transfer
- Control (branch, jump, procedure call and return, traps)
- System (OS call, VMM inst)

Instructions that manage control flow usually operate in the following ways:

1. Conditional branches
2. Jumps
3. Procedure calls
4. Procedure returns
Destination addresses are specified in each of these categories of branch except procedure return (since return address not known at compile time). Addresses are most often specified in PC-relative form to keep the code independent of where it is running. Others use register indirect jumps if the target is not known in compile time. They are useful for procedure return, case/switch statements, dynamically shared libraries, and virtual functions. Displacements are usually sign extended because branching may be in the forward as well as backward direction, although 75% of the PC-relative branches are in the forward direction. Integer programs usually have branches that are four to seven instructions away which means short displacement fields often suffice. For conditional branches, three primary techniques are in use to specify branch condition: 1) Condition Code, 2) Condition register, 3) Compare and branch.

Procedure calls have to deal with state saving; at a minimum the return address must be saved. Compilers may choose either caller saving or callee saving. Most compilers will conservatively caller save any register that may be accessed during a call.

Encoding an instruction set will usually include the following fields of information:

1. Address specifier: what addressing mode for the operand
2. Register fields
3. Opcode

The encoding affects the size of the compiled program as well as the implementation of the CPU. So the architect should balance the number of registers and addressing modes with the average program size and the instruction length. Too many operands with too many addressing
modes would require many bits since each operand requires its own address specifier. There are
variable, fixed, and hybrid types of encoding. Most RISCs including the SH3 use fixed encoding
since it is easier to decode and therefore good for performance and logic size.

As the discussion above indicates, the compiler is now the key to an instruction set architecture,
especially since most codes today are written in high level language such as C. Architects can
help the compiler writer by applying the following principles to a design:

1. Regularity. Create orthogonal instructions (operations, data types, and addressing
modes that are independent of each other)

2. Provide primitives

3. Simplify trade-offs

4. Provide instructions that bind the quantities known at compile time as constants.

1.3 Background on Pipelines:

Pipelining is the key implementation technique used to make fast CPUs, particularly RISC CPUs.
It exploits parallelism among the instructions in a sequential instruction stream and makes more
efficient use of the hardware. It does not reduce the execution time of an individual instruction
(may even increase it due to pipeline overheads such as pipeline registers and additional
multiplexers), but increases the overall instruction throughput. It is an architectural feature that is
not visible to the programmer. Pipelines are usually divided into the following stages:

1. Instruction Fetch (I)
2. Instruction Decode or Register stage (R)

3. Execution/ Effective Address Calculation (E)

4. Memory (Cache) Access/Branch Completion (C)

5. Write Back (W)

Pipelining requires the use of pipeline registers or latches, which carry both data and control from one stage to the next. The actions in the first stage (I) and the early part of second stage (R) are independent of the current instruction type; they must be independent because the instruction is not decoded until the end of the R stage.

A finite-state machine could be used to implement the control following the five-cycle structure. The overall state machine can be viewed as broken down into one state machine for each stage. For a more complex machine, microcode control could be used.

We must make sure that the overlap of instructions in the pipeline does not cause a conflict on the same resource. The basic datapath uses separate instruction and data memories, which is usually implemented with separate instruction and data caches. This eliminates the conflict for a single memory between instruction fetch stage and memory access stage. Similarly for register file, it is also used in two stages, for instruction decode where registers are read and in write back stage when a register is written to. This scheme does not cause as much conflict as writing back in both Execute and Write Back stage, but some data hazards may still arise. Another problem is branching, which changes PC in a later stage in addition to the usual PC increment.
which happens for sequential instruction flow in IF stage.

The major hurdles of pipelining are hazards which include:

1. Structural hazards which arise from resource conflicts as discussed earlier

2. Data hazards which happens when an instruction depends on the result of an earlier instruction

3. Control hazards which arise from the pipelining of branches and other instructions that changes the PC.

The simplest way to overcome hazards are to stall the pipeline whenever they are encountered
but that leads to large performance degradation. There are methods designed to minimize the number of stalls required. For structural hazard in which a single memory is shared between instruction and data, a solution would be to provide separate cache memories for each (Harvard architecture, as discussed earlier) and to use multiport register files. In the case of data hazards, forwarding (or bypassing) may be used to supply necessary data from an earlier instruction to a later one. This is done by addingmuxes (multiplexers) to ALU inputs and place the available data from more advanced stages to the muxes. The control logic is responsible for detecting the hazard by comparing the read and write addresses and the operations performed by each instruction in the flow.

Data hazards can be classified into the following categories:

- RAW(read after write): most common
- WAW: present only if write occurs in more than one stages such as write back to register file in both E and W stages.
- WAR: Occurs in out of order issues. (not in SH3 pipeline)

We rely heavily on compiler scheduling to reduce the possibility of data hazards. But certain hazards cannot be prevented even with forwarding, such as the load (read from memory) instruction which has a delay or latency that cannot be eliminated. Under such circumstances, stall will have to be used to keep data coherent. The control signals for the stalled stages should the same as a nop instruction and the stalled instruction is re-circulated through a mux into the same register for the next cycle.
To resolve control hazards we must properly handle branch instructions. We can stall the pipeline at the onset of a branch instruction until the cycle when the new PC is determined. But with a 30% branch instruction frequency, this scheme introduces too many bubbles (stalls) and greatly reduces the effectiveness of a pipeline. Alternatively we can try to find out whether the branch is take or not earlier in the pipeline AND compute the destination instruction PC earlier. This requires the comparison logic and additional adder earlier in the pipeline (such as placing an adder in R stage in addition to the ALU adder in E stage). But there still might be stalls especially in deep pipelining. To reduce pipeline branch penalties, we can 1) utilize compile-time schemes to reduce frequency of loop branches, 2) branch prediction, or 3) use delayed branch. Delayed branch scheme works by loading and executing one or more instructions that directly follow the branch instruction (in program sequence) first and then the branch destination. The slots (cycles) between the branching instruction and branching destination in which instructions are executed are called delay slots. In practice, most machines using delayed branch allow just one delay slot. Delayed branching is a programmer-visible architecture feature. The job of the compiler is to make the slotted instructions valid and successful. Of course there is a limitation as to whether we can always place a useful instruction in the delay slot. Another problem with delayed branch is it can actually reduce performance of certain powerful hardware schemes for branch prediction. There are many issues related to exception that renders delay slot scheme difficult to implement. The SH3 architecture explicitly prohibits exception between branching instruction and its delay slot. Pipelining can be further complicated by:
1. Variable instruction length and running times

2. Sophisticated Addressing modes

3. Architectures that allow writes into the instruction space (self-modifying code).

4. Implicitly set condition codes

### 1.4 Characteristics of RISC:

RISC processors usually have a limited and simple instruction set with a large number of general purpose registers, often uses compiler technology to optimize register usage, with an emphasis on optimizing the instruction pipeline. They also have the following characteristics:

1. One instruction per cycle; (more on superscalar implementations)

2. Register-to-register operations

3. Simple address modes

4. Simple instruction formats

### 1.5 SH3 Instruction Set Architecture Analysis

#### 1.5.1 Summary:

The original Hitachi SuperH RISC Engine is targeted at the embedded market and therefore has many PDA (Personal Data Assistant) functions integrated onto one chip. It has a very high code density, similar to the 68000 and x86 CPUs and about twice that of the PowerPC. Because of the small instruction size, there are no immediate load instruction, but a PC-relative addressing mode is supported to load 32 bit values. It has MAC (Multiply and Accumulate) instructions and
MACH/L (High/Low) result registers to produce 32, 42, as well as 64 bit results. It is a high-end version of the traditional integrated microcontroller and is adequate for building PDA (Personal Data Assistant) and similar devices. It is used in many of Hitachi’s own products, as well as the Sega Saturn video game system and many Windows CE palmtop computers (SH3 chipset).

The SuperH is a load-store architecture with sixteen 32-bit general purpose registers, eight banked registers, five control registers and four system registers. It uses 32-bit data and 16-bit instruction and is able to address up to 4 gigabytes of memory space. The instruction set is optimized for C language. The processor uses the popular 5-stage pipelining scheme and the programming manual suggests that write back to register file may happen in both E and W stages. But for ease the complexity of data forwarding, we will have all register writes occur in the write back cycle.

1.5.2 Analysis:
We are primarily concerned with the SH3 CPU core and its instruction set. The SH architecture brings down cost in at least two ways: the relatively simple architecture keeps the CPU core implementation small, and the instruction length is fixed at 16 bits instead of the then RISC-standard 32 bits. A fixed length instruction set facilitates simple, fast pipelines. But there are also some non RISC-like features that mostly stem from the short instruction length. RISC processors usually have large register file, SH has sixteen registers, which fall in the low end of the spectrum. Sixteen registers would require 4 bits to identify and therefore SH3 has only at most 2 register fields within an instruction. This characteristic prevents it from implementing the
three-address register-to-register operations that are prevalent in RISC architectures. Another non-RISC feature is the read-modify-write memory operations performed by some logic instructions that operate on byte length data (such as AND.B). This complication may reduce instruction throughput. Most RISCs perform only one memory reference per load or store.

Some implication of the two-address register-to-register operation is that an extra move is required if one of the source operand is to be preserved since any arithmetic instruction is bound to destroy one operand. In addition, R0 is implicitly referenced for instructions that need more than two register fields. But this strategy means the compiler would have to be more complex and needs to generate extra register-to-register move to move data into and out of R0. But the fact that R0 is a true register rather than a fixed-value register (many machines hardwire R0 to zero) could be advantageous for an architecture with such a small set of GPR.

The 16-bit instruction also implies reduced length of offsets for branches, displacements for loads and stores, and immediate fields for arithmetic and logical operations. SH3 has maximum unconditional jump offset at 12 bits, and maximum offset for conditional branches at 8 bits. The largest displacement or immediate field is also 8 bits. Addresses that do not fit this format (outside of range specifiable by the displacement or offset bits) must be precalculated in a register and then loaded. Constants longer than 8 bits must be taken from data memory using short displacement loads.

SH3 has a relatively large set of addressing modes including immediate, register indirect,
pre-increment, post-decrement, displacement, indexed, scaled, etc. The only ones that it does
not implement are direct and memory direct. For addressing modes that use a displacement
(implicit or explicit), the displacement amount is scaled by the operand size. For example, if the
operand is word size, the displacement would be x2 as shown by the following series of
instructions:

\[
\begin{align*}
\text{MOV.B } & \text{RO, @(disp, GBR) } \quad \text{RO}\rightarrow(\text{disp } + \text{GBR}) \\
\text{MOV.W } & \text{RO, @(disp, GBR) } \quad \text{RO}\rightarrow(\text{disp} x2 + \text{GBR}) \\
\text{MOV.L } & \text{RO, @(disp, GBR) } \quad \text{RO}\rightarrow(\text{disp} x4 + \text{GBR})
\end{align*}
\]

This scheme effectively expands the range addressable using displacement method.

The Hitachi keeps the unconditional branches delayed and provide the option of either delayed
or non-delayed for conditional branches (BF, BT vs. BF/S, BT/S). Most other RISC architectures
are going through a transition from delayed to non-delayed. But SH3 does provide useful
features such as "illegal slot checking" to prevent a program from having a branch within the
delay slot of another branching instruction. In addition, no interrupts or traps are allowed
between a delayed branch and its delay-slot instruction. This simplifies interrupt processing so
the hardware would not have to save two PC addresses. To improve pipeline utilization, load
scheduling may also be done by compiler but that is not mandatory.

The SH3 supports atomic multiply instructions and step divisions. There are a wide range of
multiply instructions for various operand length as well as signed and unsigned. The MAC is
unique because both of its operands are memory-based while most other architectures are register-based. SH3 has multiply-and-accumulate unit off CPU and does not have a hardware divide unit like the SH2. Therefore $DIVI$ instruction may be implemented in software.

The SH3 uses what are called System Control Instructions such as $LDC$, $LDS$, $STC$, and $STS$ instructions to load or store from the special registers (system and control registers). These tend not to be used very frequently but their implementation takes up quite a bit of opcode space.

When it comes to conditional-branch architecture, the SH3 tries to strike a balance between condition-code and compare-and-branch category. It uses what is kind of like a condition register, a T bit in the Status Register. An explicit compare instruction sets the T bit of condition code and then branches based on that. A pure condition code architecture would have all arithmetic instructions set a series of bits based on zero, carry, overflow, auxiliary, etc, and then branch based on a particular bit or a combination of those bits. Compiler must sequence the branch instruction after an arithmetic instruction for this model. A pure compare-and-branch approach would both compare and branch in one instruction and thus simplify compiler generation. The SH3 makes the compromise again based on its limited register space.

The SH3 special purpose registers are defined as follows:

**Control Registers:**

- **R0_bank - R7_bank:** Banked Registers
- **SR:** Status Register, contains condition code, saturation bit, interrupt mask level, etc.
SSR: Saved Status Register, saves SR in exception.

SPC: Saved Program Counter, saves PC in exception.

GBR: Global Base Register, points to a global data area.

VBR: Vector Base Register, points to an area where the addresses of exception handling routines are stored.

System Registers:

PC: Program Counter.

PR: Procedure Register, serves as a one-element stack for subroutine calls that saves the return address. If the called routine itself needs to do a BSR or JSR (subroutine within subroutine) then it must explicitly save (STS.L PR, @-Rn) and restore the contents of PR (LDS.L @Rm+, PR). For most calls this model is efficient because the JSR and BSR will not have to specify a register (precious!) either implicitly or explicitly.

MACH/L: Multiply-and-Accumulate unit’s operand and result registers high/low.

1.5.3 Exception Handling:

SH3 uses the following memory mapped registers for exception handling:

TRA: TRAPA exception register

EXPEVT: Exception event register

INTEVT: Interrupt event register
Usually the contents of the program counter (PC) and status register (SR) are saved in the saved program counter (SPC) and saved status register (SSR), respectively. Then execution of the exception handler is invoked from a vector address. At the completion of the exception routine, the exception handler issues \textit{RTE} (return from exception handler) instruction to restore the contents of PC and SR to the original instruction address and processor status of the point when exception first occurred.

A basic SH3 exception processing sequence consists of the following operations:

1.) The contents of the PC and SR are saved in the SPC and SSR respectively.
2.) The block (BL) bit in SR is set to 1, masking any subsequent exceptions.
3.) The mode (MD) bit in SR is set to 1 to place the SH3 in privileged mode.
4.) The register bank (RB) bit in SR is set to 1.
5.) An encoded value identifying the exception event is written to bits 11 to 0 of the exception event (EXPEVT) or interrupt event (INTEVT) register.
6.) Instruction execution jumps to the designated exception processing vector address to invoke the handler routine.

1.6 SPARC V8 Background

The SPARC V8 design in VHDL provides a good template for the SH3. SPARC, or the Scalable
Processor ARChitecture is a RISC processor specification SUN licensed to other companies. It is "scalable" mainly because the register stack can be expanded. It has 128 or 144 registers organized into windows of 24 at a time together with eight global registers. The window is moved 16 registers down the stack during a function call, so that the upper and lower 8 registers are shared between functions, to pass and return values, and 8 are local. The window is moved up on return, so registers are loaded or saved only at the top or bottom of the register stack. This allows functions to be called in as little as 1 cycle. Register 0 is hardwired to the value zero. The SPARC has some similarities with the Hitachi most notably they are both RISC architectures with similar five-stage pipelines. The ALU components such as adder, shifter, and boolean units are also very similar. Some of the major differences are the windowed register file and some special registers. In addition, SPARC uses 32-bit instruction to address 32 and 64 bit data; the SH3 uses fixed 16-bit instruction and fixed 32-bit data. SPARC uses a on chip MAC, SH3 uses a MAC off CPU, etc.
Chapter 2: Design and Implementation

2.1 SPARC datapath

The following is an examination of the SPARC datapath design. Please refer to Table X for an organization of the modules.
The **DATAPATH** module consists of the following submodules:

*Figure 2: SPARC modules organization*
Table 2: SPARC module functionalities

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTREG</td>
<td>Extracts from an instruction read address selects for use by the register file. Also outputs entire length of instruction to be pipelined into datapath as well as control logic.</td>
</tr>
<tr>
<td>REGDRIVE</td>
<td>Takes the data read from register file as input, drives them onto bus REGA and REGB. Hardwire the constant zero to mux input of RegA bus.</td>
</tr>
<tr>
<td>WDATAREG</td>
<td>Pipelines data from R stage, together with forwarded data, reorders if the data is less than 32-bit long, and output as memory write data.</td>
</tr>
<tr>
<td>BYTESWAP</td>
<td>Swap and extend data read from memory if data is less than 32-bit long before using in registers or forwarded.</td>
</tr>
<tr>
<td>RESULTREG</td>
<td>Pipelines result data from E stage to be output to register file and forwards these data to reg_a and reg_b in R stage.</td>
</tr>
<tr>
<td>EREGS</td>
<td>Drives R stage RegA/B data, immediate/displacement data, and forwarded E_result to the pipeline registers E-A and E-B.</td>
</tr>
<tr>
<td>SHIFT</td>
<td>Barrel shifter that shifts the 32-bit A input by the amount specified by the 5 bit B input.</td>
</tr>
<tr>
<td>BOOLE</td>
<td>The boolean unit performs the 16 possible 2-input logic operations, e.g., AND, OR, XOR, ZERO generate, etc on a pair of inputs A and B.</td>
</tr>
<tr>
<td>YREG</td>
<td>Multiply/Divide Register. Contains the most significant word of the double-precision product of an integer multiplication, as a result of either an integer multiply instruction, or of a routine that uses integer multiply step instruction. The Y register also holds the most significant word of the double-precision dividend for an integer divide instruction.</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply-and-Accumulate unit performs multiplication through multiple cycle calculations.</td>
</tr>
<tr>
<td>ADDER</td>
<td>ALU full adder performs add, subtract, zero, negative, overflow detection. Also outputs memory address.</td>
</tr>
<tr>
<td>ASR</td>
<td>Ancillary State Register. For use by software to read/write unique implementation-dependent processor registers. There are up to 32 of them. They are implementation-dependent such as timers, counters, diagnostic registers, self-test registers, and trap-control registers. A particular integer unit may choose to implement from zero to sixteen of these ASR's.</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter. Calculates the next instruction address under normal flow as well as branching, jumping, and subroutine calling. Outputs instruction and data address to memory. Provides PC read back.</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>REGFIL</td>
<td>Register file module. Two read port and one write port. Depending on the MSB of the register select address, the data is read either from the global register or the regular register file.</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Control logic module. Consists of the following sub-modules.</td>
</tr>
<tr>
<td>LDST_</td>
<td>Load-store control to and from memory</td>
</tr>
<tr>
<td>CONTROL</td>
<td></td>
</tr>
<tr>
<td>REGCTL</td>
<td>Register control. Controls bypassing onto the R stage bus REGA and REGB</td>
</tr>
<tr>
<td>TRAP</td>
<td>Trap logic</td>
</tr>
<tr>
<td>PSR</td>
<td>Program Status Register logic. Certain fields of the register are implementation specific.</td>
</tr>
<tr>
<td>RESULT_</td>
<td>Concerns results from various ALU computations</td>
</tr>
<tr>
<td>CONTROL</td>
<td></td>
</tr>
<tr>
<td>FSM</td>
<td>Finite state machine logic control for different processing states</td>
</tr>
</tbody>
</table>

2.2 **Hitachi Datapath**

The SH3 CPU core will take up about 3mm\(^2\) on a 0.5\(\mu\)m process assuming a layout density of 20,000 transistors/mm\(^2\).

2.2.1 **Datapath Summary:**
Figure 3: SH3 datapath
Everything within the datapath belong to one of five pipeline stages. The Hitachi datapath can be viewed as having the following several parallel paths:

1. The path (green) that provides memory address from adder in E stage and subsequently loads memory read data (Mdata_in) and send to byteswap in W stage (W_result), providing data for RF and SPR(Special Purpose Registers, including System, Control and Banked Registers). This path is defined in \textit{h_add}, \textit{h_addr}, and \textit{h_byteswap}.

2. The path (light orange) that takes E stage values, EA, EB and Eresult, pipelines to W stage (swp_data) and provides again for RF and SPR. This path is defined in \textit{h_rsltrg}.

3. The path (blue) that takes R stage data (RegB, RegC) together with forwarded E_result, reorders the data, then pipelines to E and C stage, providing memory write data. This path is defined in \textit{h_wdata}.

4. The PC path (pink) which simply pipelines the I stage PC all the way to W stage supplying forwarding data and SPR write data. This path defined in \textit{h_pc}.

5. The IPC generate path (yellow). This is the path that computes the next instruction in pipeline. It is parallel to the decode and execution of the current instruction. For sequential flow, the current instruction address (IPC) is incremented by four in I cycle to generate the next address to be fetched in R cycle. For branching, the destination address is determined in the R cycle and instruction fetched in E cycle. This path also defined in \textit{h_pc}. 
Data sources and destination: Disregarding the \textit{h_address} module, we can view the datath path as having separate instruction and data memory (Harvard architecture). Thus there is instruction memory in the IF stage as well as data memory in the WB stage. The instruction memory is read-only where as the data memory is both read- and write-able. In addition, there is the general purpose register file in R stage which is read-/write-able. Some system and control registers residing in the \textit{h_pc} module are also read-/write-able.

Source: InstMem, GPR, DataMem, SPR
Destination: GPR, DataMem, SPR, PC

2.2.2 Placement of Registers:

Organization-wise banked registers may be placed in the same place as the general purpose registers in the Register File (\textit{h_regfil}). Banked registers are used for ease of context switching in cases of interrupts and exceptions. GBR, VBR, SSR, SPC and PR may all be placed in the PC module for easy access. SR is closely related to control logic and can be stored in control, so it does not make up a physical datapath register. The MAC registers (MACH and MACL) reside on a separate MAC unit which is off CPU. All registers are read in R stage and written to in W stage except SR and MAC. SR may be read in R stage normally just like most other control registers. It should be written to in E stage, however, in order to minimize the number of instructions to be stalled (Since SR is simultaneously visible by control logic for all pipeline stages). MAC registers are accessed like data memory and are both read and written to in memory cycles. (Future work: The interface is not well defined yet but if no extra signal is used
to interface datapath to MAC then there may be contention between MAC and memory access, then need to split into extra cycles.)

PC (Program Counter) register can be considered as providing data in the I stage (from I_PC) and written to in R or E stage relative to the current instruction. In the case of most other data sources, the data is available at the end of the data access cycle; for the PC module, however, the next address is not available for the PC until the cycle after the computation is done. For example, I_PC is incremented by four in I stage, but this address is not used until R stage. A branch instruction on the other hand will computed the branch destination address in R stage and that address will be used in E stage to fetch the new instruction.

Instructions can be categorized according to the path they take (source and destination). An instruction can often take multiple paths simultaneously. They can be roughly divided as follows, with load and store defined in reference to memory:

1. Memory Store, in which mem is the destination
2. Register Load, in which mem is the source and Reg is the destination
3. Register transfer, in which register is the destination, source could be register, imm/disp, or ALU computation results
4. Branching, which deals with PC and alters flow
5. Others including those that updates SR, MAC, and miscellaneous instruction

For example, the autoincrement/decrement instruction MOV.B Rm, @-Rn does the following:
Therefore, this instruction will be considered as both memory store and register transfer.

Most of the system control instructions like STC/S and LDC/s are most often register transfer instructions. The auto-increment and decrement instructions are memory store or register load in combination with register transfer of ALU results (inc or dec).

The data transfer instructions (MOV) are categorized similarly; the ones that have combination categories are usually auto-inc/dec instructions. The displacement and index addressing mode instructions requires ALU computation to derive the memory address.

Most of the arithmetic instructions are register transfer instructions that places ALU results in registers. Some of the compare instructions only update SR. Some of the logic instructions such as AND.B are atomic read-modify-write instructions that operate on memory read data and then write back to memory. These together with the MAC related instructions may be considered as category 5.

<table>
<thead>
<tr>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegA &lt;= Rm</td>
<td>Edata &lt;= RegA Eresult = Rn -1</td>
<td>Mdata &lt;= Edata E_addr = Eresult Write mem</td>
<td>Rn=Rn-1 Write RF</td>
</tr>
<tr>
<td>RegB &lt;= Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The branch instructions are naturally category 4.

Writing to SR takes three EX cycles, and no new instructions can be dispatched during the entire pipeline until the last EX cycle is over. This is indicated in the appendix of the programming manual on pipeline. For example:

$LDC\ Rm,\ SR$ instruction pipeline:

| Time Slot | t        | t+1 | t+2 | t+3 | t+4 | t+5 | t+6 | t+7 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC</td>
<td>I</td>
<td>R</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next Inst</td>
<td>I</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3rd Inst</td>
<td>I</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The most important computation resource is the ALU, which provides to the datapath E_result (from various ALU modules) and E_addr (which is the data memory read/write address computed by the carry-select adder). In addition, there is an incrementer in I stage of PC module that computes exclusively the increment by 4 value of the next PC address. There is a more versatile carry look ahead adder in R stage of PC which computes the branch destination address. Judging by the programming manual, the original SH3 design probably does not have this extra adder. The design seems to make branch instruction wait until the E stage to share the use of the ALU adder to compute destination address, in which case the branch destination instruction will not be fetched until the C stage of the branching instruction. For example, the pipeline of an unconditional branch instruction such as $BRA$ is indicated to be:
The operation of BRA is explained in pseudo C code as follows:

```c
BRA(long d) /* BRA disp */
{
    unsigned long temp;
    long disp;
    if ((d&0x800)==0) disp=(0x00000FFF & d);
    else disp=(0xFFFFF000 | d);
    temp=PC;
    PC=PC+(disp<<1)+4;
    Delay_Slot(temp+2);
}
```

The branch destination instruction is not fetched until after E stage of the BRA instruction. So it is very likely that the destination address is calculated in E stage of BRA, especially since the PC used in the addition is I_PC+4 and the displacement refers to the distance from branch destination address to “Next Inst” rather than BRA.

Rather than stalling the delay slot and subsequent instruction or fetching and flushing the second delay slot, we can increase throughput by placing an extra adder in PC module exclusively for the purpose of computing branch destination address in R stage. Then the pipeline of the BRA instruction can be modified to the following:

---

1That is actually E_pc when the branching instruction is at E stage. But we do not want to use that since it complicates the situation for interrupt or exception, although exception is not allowed between a delayed branch and its delay-slot instruction.
The operation is the same in pseudo C code because there should be no programmer-visible differences. This implies that the displacement calculation will be kept the same.

Most of the pipeline registers have a control line called ADVANCE that controls whether the old value should be recycled or new value should be advanced. This is implemented mainly to deal with the stalls caused by memory latency in memory LOAD type instructions.

-Forwarding:

(-Mux intro With digital design, mux can be implemented either with pure AND OR logic gates, with transmission gates, or with shared tri-state bus. Explain each But in any case, some times more than one level of mux is needed to reduce the load??) 

Most forwarding is done by driving data from other stages to R stage busses. Within R stage there are two levels of busses that act as multiplexers. First there are RegA, RegB, RegC, and RegBus (PC), then there are EA, EB, and Ewdata. Most forwarded signals go to the first level since they are usually obtainable early within their source stages. E_result is forwarded to the
later stage because it is obtainable only from at the very end of E stage.

Alternatively, forwarding can be done at the beginning of E stage instead of at the end of R stage. But the ALU operations in E stage is already a lengthy process. In order not to increase the critical path, we choose to forward to R.

Instructions that take multiple cycles, especially those that operate on memory read data (such as the atomic read-modify-write instructions), can also take advantage of forwarding path to send their data to ALU, and from ALU to the write datapath.

Any R stage signal whose source might be modified by a signal in the more advanced stage should have the modifying signal forwarded. Signals that write to register files (both general purpose registers and banked registers), system and control registers, data memory, and program counter all should have should be forwarded. It should be noted that memory read data (mdata_in) is not forwarded in C stage because the data HAVE to go through reordering in BYTESWAP which is in W stage.

2.2.3 Hitachi Design

The following changes will be made to the SPARC design in VHDL code to enable the Hitachi ISA:
First of all, note that SPARC has three register operand fields while the Hitachi only has two. So the Hitachi effectively has to multiplex one of the fields for both register read and write address. We can keep the register file busy reading much like we would keep the memory busy reading by always supplying two register read address: Rs0 for bits 7 through 4 and Rs1 for bits 11 through 8 even if the actual instruction may or may not have these two source operands. Later down the pipeline control signal will select/ignore the appropriate data to load to the various busses as needed. In addition, the INSTREG module will try to detect instructions \textit{STC Rx\_bank, Rn} and \textit{STCL Rx\_bank, @-Rn} to see if Rs0 (bits 7 - 4) refers to a banked register as source. If so, then the MSB of Rs0 field will be ignored in forming the register address select. For example, 1011 would specify the register 011 which is R3\_bank. These information are supplied to register file
directly from datapath to reduce critical path. But the control logic still has to provide register file information as to which bank to select. This is relatively straightforward because the RB bit of SR register will work. The register addresses are passed as bit vectors and converted to natural values within the REGFIL module. Rm and Rn as referred to in the programming manual do not have a direct correspondence to the register fields of Rs0 and Rs1. The matching of Rm and Rn to a data bus will be done when data is read from register file in REGDRV module.

Instructions and data are both always fetched in 32-bit words which is two instruction words long. Therefore, instruction fetch only has to happen every other cycle. On the non-fetching cycle, the instruction word fetched in the previous cycle is fed back but a different half will be selected to be the current instruction. Among the outputs this module INSTREG generates, both $q$ and $q_{\text{bar}}$ are the entire instruction. $q$ is for use by other datapath elements whereas $q_{\text{bar}}$ is the inverted version (to maintain signal strength) outputted to the control logic (which is outside of datapath and more distant). Two bit-length outputs, $rinst11_{\text{bar}}$ and $rinst7_{\text{bar}}$ are generated to control logic to provide sign extensions for decoding immediate/displacement data.

2.2.3.2 REGDRV:

Register Read data 0 (Rd0 and Rd1) are driven to both RegA and RegB so that RegA can be made corresponding to Rn and RegB to Rm. Register zero is driven only to RegC. Some constants are driven onto the Reg busses. In particular, 0x160 is hardwired to RegB for use in TRAPA instruction, and SR is driven here from the control logic. (The exact form of SR depends on the control logic because SR does not come from the datapath)
2.2.3.3 WDATA:

This is the datapath that pipelines data from R stage to C and provides mdata_out to be written to memory. Two sources, regb, and regC, together with the forwarded E_result are driven onto the e_wdata bus. At E stage, instruction TAS.B might set bit 7 of the write data to 1. E_wdata then goes through some reordering, the purpose of which is to copy byte and word length data over the whole 32 bits width of the bus so that memory access can still be done on 32 bit or quad address boundaries. This reordering ensures that write logic only has to selectively enable the proper address(es) when committing the write. The complement of this is the ByteSwap logic done for data read from memory (mdata_in). For read-modify-write operation, e_result is forwarded to end of R stage and then goes through E stage before going to memory in C stage. This is to reduce the complexity of the datapath at the cost of maybe one extra cycle. Since
read-modify-writes are rarely used and load-store architecture do not usually transfer ALU results directly to memory this tradeoff is probably worthwhile. For read-modify-write operations, instead of having the pipeline I-R-E-C-E-C as suggested by the programming manual, it would be I-R-E-C-W-E-E-C.

**Figure 6:** WDATA module

**BYTESWAP:**

No change except going from 64-bit to 32-bit. Takes mdata read from memory in C stage, pipelines into W stage. The byte and word length data are then rearranged so they are aligned along the least significant bits of the 32-bit word boundary and the high order bits are properly extended. This is the complement operation of the reordering that happens in WDATA module before the write data is sent to memory. Byte swap happens in W stage to reduce critical path in C stage since reading from memory is not a particularly fast process. Consequently, bypassing of memory read data is usually done from W stage rather than C stage. There might be more cycles in operations involving memory read than what the manual noted. For atomic read-modify-write operations, instead of pipelining at I-R-E-C-E-C, it will be I-R-E-C-W-E-E-C
2.2.3.4 RESULTREG:

This module has three functions. First it provides data for register writes, including general purpose registers as well as special registers. It outputs two data bus, w_result0 and w_result1, in case of simultaneous register writes. Both busses are fed to the two write ports of the register file, but only w_result1 is capable of writing to banked registers. w_result1 is also fed to PC module for special register writes. Most instructions only write to one register destination at a time. Even the instructions that do write to different registers/special registers simultaneously do not ever have the same source for those data. This module also performs immediate/displacement data extraction from the r_inst data and drives that to RegA, B, and C. Finally, RESULTREG drives bypassing data from various stage onto RegA and RegB bus.

Figure 7: RESULTREG module
For register write the data sources are: EA, EB, mixed EA/EB (for instruction XTRCT and SWAP), E_result, and swp_data (memory read). For forwarding the data source are: swp_data (memory read), C_result and W_result from the register write pipeline.

2.2.3.5 EREGS:

This module pipelines R stage data busses to E stage ALU operands: RegA to EA and RegB to EB. REGC is muxed onto both EA and EB. Some constants useful for ALU operations (for specifying shift amount and increment by one, two, or four) are muxed onto both pipeline registers. E_result is forwarded here at the end of R stage to both pipeline registers because its source is not available until the end of E stage.

Figure 8: EREGS module
2.2.3.6 SHIFT:

The barrel shifter shifts in through sin bit either T bit, zero, MSB or LSB of operand A. After shifting left or right by the amount specified in the LSB five bits of B operand, it outputs the 32-bit result as well as the output bit sout. It shifts according to the amount bits in the following order: 4,0,3,1,2 for both left shift and right shift.

2.2.3.6 BOOLE:

In addition to performing the regular boolean operations, this module also extends an operand (for EXTS/U instruction) and does string compare (CMP/STR instruction). The extension operation is done by using a constant mask to set or clear the high order bits to be extended. For example, the operation of EXTS.B may be explained in pseudo C code as:

```
EXTSB(long m, long n) /* EXTS.B Rm, Rn */
{
    R[n]=R[m];
    if ((R[m]&0x00000080)==0) R[n]&=0x000000FF;
    else R[n]|=0xFFFFFFFF;
    PC+=2;
}
```

So depending on bit 7 of Rm, either 0x0000000FF will be ANDed to Rm or 0xFFFFFFFF will be ORed to Rm. The string compare is done by comparing the four bytes simultaneously and Oring the results. The boolean operations are implemented using four OP.

\[
F = (O0 \cdot A' + O1 \cdot A) \cdot B' + (O2 \cdot A' + O3 \cdot A) \cdot B'
\]
2.2.3.8 MAC:

On the SH3 chip, the 32-bit MAC is an off CPU unit. This may be left to future work section where it can be implemented as a multiplier with adder or a multiplier by itself that utilizes the adder of ALU. MACL/H registers reside on the MAC unit. Therefore to access MAC for read or write the CPU has to go through the memory cycle.

2.2.3.9 PC

This module has three functions:

1. Instruction address sequencing, PC
2. Special register access
3. E_address pipelining

The special registers GBR, SSR, SPC, PR and VBR are all placed in PC. They use w_resultl data from RESULTREG module or w_pc data from within the PC module as source. They are collectively output to bus reg_bus, which is then fed to addend1 of the adder whose sum output supplies the branching address to PC. Addend2 are either various constants or NPC. Reg_bus is also driven to RegB to provide data from PC module to the ALU. RegB, in turn, drives data from the rest of the datapath onto reg_bus. They provide each other with data including forwarding data. This relies on the fact that no two instruction can simultaneously operate in any R stage (or any other stage for that matter). I_PC is output as instruction address in Instruction Fetch cycle. It is simultaneous incremented by four in I cycle for use on the next cycle if program flow is sequential. I_PC is also pipelined through all subsequent stages to write special registers in W stage. From various stages, PC is forwarded to reg_bus because certain
Table 3: BOOLE OP bits

<table>
<thead>
<tr>
<th>OP Bits</th>
<th>Boolean Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1</td>
<td>AND</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>ANDN</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>OR</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>ORN</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>XOR</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>XORN</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>ZERO</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>B</td>
</tr>
</tbody>
</table>

2.2.3.7 ADDER:

Aside from performing addition in carry select, the ADDER module also provides data address to memory. The address is taken from both addend, A and B, and their sum. In addition, a few constant address of exception registers are added. Output B31 is useful for \textit{DIVOS} instruction. Input A can be shifted left by one bit with a shift in bit that is useful for \textit{DIV1} instruction.

Figure 10: ADDER module
Figure 11: PC module

Instructions (STC, STS) may read the special registers in the delay slot of certain branch instructions (BSR, BSRF, JSR, etc).

The other function of the PC unit is to pipeline the memory address, e_addr, computed by the ALU adder in E stage to Daddr in the C stage.

As explained earlier, there is an incrementer that computes I_PC+4 in I stage and an adder that
computes branch destination address in R. A branching instruction will start fetching the
destination address in E stage of the branching instruction. For those with delay slots, delay slot
instruction is fetched in the R stage of the branching instruction. If the branching is a conditional
one, the next instruction in sequence is fetched in R stage, but depending on whether the branch
is taken, this instruction either proceeds or is flushed in E stage (of the branching instruction). If
the branch is not taken, the second next instruction in the sequential flow will be fetched in E
stage of the conditional branching instruction. For example, the conditional branch instruction
BF disp will have the following pipeline if condition is satisfied:

| Time Slot | t | t+1 | t+2 | t+3 | t+4 | t+5 | t+6 | t+7 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BF disp</td>
<td>I</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Next Inst | I |     |     | I   |     |     |     |     | (Fetched but discarded/flushed)
| Branch dest | I | R   | E   |     |     |     |     |     |     |
| ...       |   |     |     | I   | R   | E   |     |     |     |

If the condition is not satisfied (branch not taken):

| Time Slot | t | t+1 | t+2 | t+3 | t+4 | t+5 | t+6 | t+7 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BF disp</td>
<td>I</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next Inst</td>
<td>I</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch dest</td>
<td>I</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>I</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2.3.10 ADDRESS

This is an extra module that merges the instruction address and the data address to a single memory address. For each instruction or data address, control logic selects the one to output to memory for that time slot. If an address is not selected, it is cycled through a register to become saved_xadd and possibly be selected in the next cycle. The rest of the datapath modules may view the system as having separate instruction and data memory.

\[\text{Saved_iaddr}\]
\[\text{I_addr}\]
\[\text{Addr_out}\]
\[\text{Saved_maddr}\]
\[\text{M_addr}\]

Figure 12: ADDRESS module

2.2.3.11 REGFILE

The register file is not divided into register windows, but it now contains both general purpose registers and banked registers. It has two read address select (but three data readout ports) and two write ports. The third read port always outputs data from Register Zero. Only one read port
Figure 13: REGFILE module

(rd0) may access the banked registers. Similarly, only one write port(wradd1) may write to banked registers and it is only sometimes enabled because most instructions only write one registers at a time.
There would be a conflict if both writes are trying to access the same physical register. For the read port that has access to the banked registers, signal rd0type indicates whether the address selects a general purpose register (4 bit address) or a banked register (3 bit address). Signal rd0bank holds information as to which bank the addressed is for, bank0 or bank1, as indicated by RB bit of the Status Register. The write counterparts to these signals are wr1type and wr1bank.

2.2.4 Tools:

The datapath and register file will be written in VHDL. VHDL is the VHSIC (Very High Scale Integrated Circuit) Hardware Description Language that is capable of describing various levels of abstraction from behavioral down to gate and switch level. The model described can then be compiled and simulated at those different levels in order to verify their functional operation and performance parameters. The VHDL file can be synthesized into logic blocks and optimized for a specific technology using its cell library. We are mostly writing RTL (register transfer language) models of the datapath and control logic. There are some behavioral models for the test bench. One advantage of VHDL is that various architectures can be written simultaneously for the same module. This enables mutual design checking and different levels of abstraction. The simulation software used is called ModelSim by ModelTech.
Chapter 3: Future Work

Optimize the datapath. First of all, a compiler writer should be consulted both to improve the design and to reach a clear understanding on the interpretation of the programming manual. Code optimization based on the instruction pipelines as outlined by the manual should probably be avoided since our implementation only conforms to the SH3 architecture in programmer-visible features. In particular, the delay slot timing is different from the manual and the way displacement is calculated should be discussed. The compiler writer may also advise on the feasibility of certain hardware features. For example, there are a lot of forwarding of data in the datapath, some more often used than others. With each data bus 32-bits wide, would it make good sense to stall the pipeline instead of forwarding the data for certain non-frequent hazard situations? Regarding the register file, should write back from E stage allowed in addition to W stage? The compiler writer will have a better estimate of how often certain code combinations may arise. We also need a good estimate of the final processor and system area to assess whether an extra adder in R stage of PC module is affordable.

The MAC unit needs to be implemented. It is recommended that it be implemented as a 32x8 Booth multiplier to save on area. If it is implemented off CPU, then the comment on accessing MACH and MACL in memory cycle would apply. Otherwise it could be implemented as part of the ALU and utilizing the ALU adder, in which case care should be taken to prevent MAC from hogging execution cycles.
Once the datapath is determined, the corresponding control logic should be implemented. First generate control for individual instructions. The instruction pipeline table in the appendix may help in doing that. The control logic should take care in generating control signals for tristate controlled muxes and busses so that only one source may drive the bus at a time and therefore only one control signal can be active at a time. Then the control logic for detecting data hazard should be implemented and the proper forwarding signal be enabled; or if necessary, the pipeline should be stalled and the control logic will emulatenop on stalled stages. To resolve control hazards (related to branching), we need to determine when an instruction already fetched into the pipeline should be flushed. The Status Register (SR) should be integrated into the control logic design. Also needed are exception handling capabilities.

Once the design is completely coded, then simulation can start as explained earlier. The datapath code may be hand translated to gate level digital circuit because first of all the logic is largely combinational and secondly it is important to minimize its critical path. The control logic code will be synthesized using software tools to generate the circuits.

3.1 Testing and Verification

VHDL modules may be simulated and tested individually. Test signals for the module under test can be generated either through a test bench or by executing a macro file in the simulator describing the signals. Test bench can also written in VHDL and is useful for complicated modules because of the bench can capture the results, output them to a file and compare them with a golden result (pre-determined correct result). The whole procedure may be automated.
Initially the functional submodules such as shifter, boolean unit, adder, multiplier, and register file will be simulated and tested. Once the entire chip (datapath and control) is captured in RTL then an assembler can be written to use the instruction set of the processor. At this point, the assembled code can be run on the simulated chip to verify functionality.
Appendix

4.1 Instruction pipeline table
<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>STC SR, Rn</td>
<td>SR</td>
<td>RegB</td>
<td>EB</td>
<td>C_result, write Rn</td>
</tr>
<tr>
<td>(See SR instructions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STC/S SPR, Rn</td>
<td>SPR</td>
<td>RegB</td>
<td>EB</td>
<td>C_result, write Rn</td>
</tr>
<tr>
<td>STC Rx_BANK, Rn</td>
<td>or Rhank</td>
<td>or RegB</td>
<td>EB</td>
<td>C_result, Write Rn</td>
</tr>
<tr>
<td>STC L SR, @-Rn</td>
<td>Rn</td>
<td>SR</td>
<td>RegA 4</td>
<td>E addr</td>
</tr>
<tr>
<td>(See SR instructions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STC/S L SPR, @-Rn</td>
<td>Rn</td>
<td>SPR</td>
<td>RegA 4</td>
<td>E addr</td>
</tr>
<tr>
<td>STC L Rx_BANK, @-Rn</td>
<td>Rn</td>
<td>Rhank</td>
<td>RegA 4</td>
<td>E addr</td>
</tr>
<tr>
<td>LDC Rm, SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(See SR)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDC/S Rm, SPR</td>
<td>Rm</td>
<td>RegB</td>
<td>EB</td>
<td>c_result</td>
</tr>
<tr>
<td>Write SPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDC Rm, Rx_bank</td>
<td>Rm</td>
<td>RegB</td>
<td>EB</td>
<td>c_result</td>
</tr>
<tr>
<td>Write Rx_bank</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDC L @Rm+, SR</td>
<td>(see SR)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction:</td>
<td>R</td>
<td>E</td>
<td>C</td>
<td>W</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>RegA</td>
<td>RegB</td>
<td>RegC</td>
<td>EA</td>
</tr>
<tr>
<td>LDC/S L @Rm+, SPR</td>
<td>Rm</td>
<td>4</td>
<td>RegB</td>
<td>Ea+Eb</td>
</tr>
<tr>
<td>LDC L @Rm+, Rx/func</td>
<td>Rm</td>
<td>4</td>
<td>RegB</td>
<td>Ea+Eb</td>
</tr>
<tr>
<td>TRAPA #nn</td>
<td>Imm</td>
<td>RegA</td>
<td>TRA_addr</td>
<td>Ea</td>
</tr>
<tr>
<td>(R after last R)</td>
<td>SR</td>
<td>RegB</td>
<td>EB</td>
<td></td>
</tr>
<tr>
<td>(E after last W)</td>
<td>0x160</td>
<td>RegB</td>
<td>EB</td>
<td></td>
</tr>
<tr>
<td>(R after last E)</td>
<td>Addr1 = RegBus = VBR</td>
<td>IPCT=AltPC=Add1 + Add2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr2=0x100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLRMAC</td>
<td></td>
<td></td>
<td>EB</td>
<td></td>
</tr>
<tr>
<td>(See MAC instructions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLRS (See SR - reg)</td>
<td></td>
<td></td>
<td>EB</td>
<td></td>
</tr>
<tr>
<td>CLRT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETS</td>
<td></td>
<td></td>
<td>EB</td>
<td></td>
</tr>
<tr>
<td>SETT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTE (See branching)</td>
<td></td>
<td></td>
<td>EB</td>
<td></td>
</tr>
<tr>
<td>Instructions not implemented</td>
<td>LDTLB, PREF @Rn, SLEEP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Data Transfer Inst

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV #imm, Rn</td>
<td>mm (ext)</td>
<td>RegB</td>
<td>EB</td>
<td>C_result write Rn</td>
</tr>
<tr>
<td>MOV.x Rm, @R0, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegC</td>
</tr>
<tr>
<td>MOV.x Rm, @-Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td></td>
</tr>
<tr>
<td>MOV x Rn, @Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td></td>
</tr>
<tr>
<td>MOV x R0, @(disp, Rn)</td>
<td>Rn</td>
<td>disp</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>MOV.L Rm, @(disp, Rn)</td>
<td>Rn</td>
<td>disp</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>MOV x R0, @(disp, GBR)</td>
<td>disp</td>
<td>GBR</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>MOV x @(disp, PC), Rn</td>
<td>disp</td>
<td>R_PC</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>MOV x @(disp, Rn), R0</td>
<td>disp</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>MOV x @(Rm+4), Rn</td>
<td>disp</td>
<td>Rm</td>
<td>RegB</td>
<td></td>
</tr>
<tr>
<td>MOV x @(R0, Rm), Rn</td>
<td>disp</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>MOV x @(disp, GBR), R0</td>
<td>disp</td>
<td>GBR</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>Instruction</td>
<td>R</td>
<td>E</td>
<td>C</td>
<td>W</td>
</tr>
<tr>
<td>----------------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>MOV x @Rm, Rn</td>
<td>Rm</td>
<td>RegB</td>
<td>EB</td>
<td>E_addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVA @(damp, PC), R0</td>
<td>dmp</td>
<td>R_PC</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVT Rn</td>
<td>0</td>
<td>1</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td>SWAP x Rm, Rn</td>
<td>Rm</td>
<td>RegB</td>
<td>EA</td>
<td>swapped</td>
</tr>
<tr>
<td>XTRACT Rm, Rn</td>
<td>Rm</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Arithmetic Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD #imm, Rn</td>
<td>Rn</td>
<td>imm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDC Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDCV Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP/EQ #imm, R0 (Z)</td>
<td>mm</td>
<td>R0</td>
<td>RegA</td>
<td>RegC</td>
</tr>
<tr>
<td>CMP/EQ Rm, Rn (Z)</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
</tbody>
</table>

58
<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
<th>W_result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RegA</td>
<td>RegB</td>
<td>RegC</td>
<td>EA</td>
<td>EB</td>
</tr>
<tr>
<td>CMP/H S Rm, Rn (V, Z)</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB</td>
</tr>
<tr>
<td>CMP/G E Rm, Rn (V, Z)</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB</td>
</tr>
<tr>
<td>CMP/H I Rm, Rn (V)</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB</td>
</tr>
<tr>
<td>CMP/G T Rm, Rn (V)</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB</td>
</tr>
<tr>
<td>CMP/P Z R Rm (Z, V)</td>
<td>Rn</td>
<td></td>
<td>RegA</td>
<td></td>
<td>EA-0</td>
</tr>
<tr>
<td>CMP/P L Rn (V)</td>
<td>Rn</td>
<td></td>
<td>RegA</td>
<td></td>
<td>EA-0</td>
</tr>
<tr>
<td>CMP/STR Rm, Rn (E)</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>BOOLE</td>
</tr>
<tr>
<td>DIV/1 Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>(EAsc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T+/-EB</td>
</tr>
<tr>
<td>DIV/S Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>Use ADD</td>
</tr>
<tr>
<td>(update Q, M, T)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV/U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT Rn</td>
<td>Rn</td>
<td></td>
<td>RegA</td>
<td>1</td>
<td>EA-EB</td>
</tr>
<tr>
<td>EXT/S U x Rn, Rn</td>
<td>Rm</td>
<td></td>
<td>RegB</td>
<td>BOOLE</td>
<td></td>
</tr>
<tr>
<td>(Use Boole to extend)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG Rm, Rn</td>
<td>0</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB</td>
</tr>
<tr>
<td>NEG C Rm, Rn</td>
<td>0</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB-T</td>
</tr>
<tr>
<td>Instruction</td>
<td>R</td>
<td>E</td>
<td>RegA</td>
<td>RegB</td>
<td>EA</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>SUB Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB</td>
</tr>
<tr>
<td>SUBC Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB-T</td>
</tr>
<tr>
<td>SUBV Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA-EB</td>
</tr>
</tbody>
</table>

**LOGIC Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>E</th>
<th>RegA</th>
<th>RegB</th>
<th>EA</th>
<th>EB</th>
<th>Errlts</th>
<th>E.addr</th>
<th>C纭</th>
<th>Meml</th>
<th>Mem2</th>
<th>W</th>
<th>W纭1</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND Rm, Rn</td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
<td>EA &amp; EB</td>
<td></td>
<td></td>
<td></td>
<td>Eresult</td>
<td></td>
<td></td>
<td></td>
<td>C_result</td>
</tr>
<tr>
<td>AND #imm, RO</td>
<td>#mm</td>
<td>RO</td>
<td>RegA</td>
<td>RegB</td>
<td>EA &amp; EB</td>
<td></td>
<td></td>
<td></td>
<td>Eresult</td>
<td></td>
<td></td>
<td></td>
<td>C_result</td>
</tr>
<tr>
<td>AND B #imm, @(RO, GBR)</td>
<td>#mm</td>
<td>GBR</td>
<td>RO</td>
<td>RegC</td>
<td>RegB</td>
<td>EA-EB</td>
<td>Eresult</td>
<td>Eaddr</td>
<td>Read ()</td>
<td>ByteSwap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(R starts same time as W of above line)

(E starts after E of above)

Similar pattern for inst OR, XOR, and TST (except no register write or memory store)

<p>| Instruction | R     | E     | RegA | RegB | EA &amp; EB|     |        |        | Eresult |      |      |      | C_result | Write Rn |</p>
<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OR #imm, RO</strong></td>
<td>Imm</td>
<td>R0</td>
<td>RegA</td>
<td>EA</td>
</tr>
<tr>
<td><strong>OR.B #imm, @(RO, GBR)</strong></td>
<td>Imm</td>
<td>GBR</td>
<td>R0</td>
<td>RegC</td>
</tr>
<tr>
<td><strong>XOR Rm, Rn</strong></td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td><strong>XOR.B #imm, @(RO, GBR)</strong></td>
<td>Imm</td>
<td>GBR</td>
<td>R0</td>
<td>RegC</td>
</tr>
<tr>
<td><strong>TST Rm, Rn</strong></td>
<td>Rn</td>
<td>Rm</td>
<td>RegA</td>
<td>RegB</td>
</tr>
<tr>
<td><strong>TST #imm, RO</strong></td>
<td>Imm</td>
<td>R0</td>
<td>RegA</td>
<td>RegC</td>
</tr>
<tr>
<td><strong>TST B #imm, @(RO, GBR)</strong></td>
<td>Imm</td>
<td>GBR</td>
<td>R0</td>
<td>RegC</td>
</tr>
</tbody>
</table>

(R starts same time as W of above line)

(R starts after E of above)
<table>
<thead>
<tr>
<th>Instruction:</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Special Instructions

- **LDC Rm, SR**
  - Rm
  - RegB sent to control logic to update SR in E stage
<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC L @Rm+, SR</td>
<td>Rm</td>
<td>4</td>
<td>Eresult</td>
<td>Read ()</td>
</tr>
<tr>
<td>CLRMAC</td>
<td>0</td>
<td>RegA</td>
<td>EA</td>
<td></td>
</tr>
<tr>
<td>STS MACH/L, Rn</td>
<td>Rn</td>
<td>RegA</td>
<td>4</td>
<td>Eresult</td>
</tr>
<tr>
<td>MULS/U W Rn, Rn</td>
<td>Rn</td>
<td>RegA</td>
<td>RegB</td>
<td>EA</td>
</tr>
<tr>
<td>MAC.s @Rm+, @Rn+</td>
<td>Rn</td>
<td>RegA</td>
<td>4</td>
<td>Eresult</td>
</tr>
<tr>
<td>MAC operation depends on whether MAC unit has its own adder</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table Columns:****
R: Register
E: Extension
C: Condition
W: Write
<table>
<thead>
<tr>
<th>Branch Instructions</th>
<th>I</th>
<th>R</th>
<th>E</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF/T disp</td>
<td>IPC+4</td>
<td>NPC=IPC+4</td>
<td>l/T=0/1, next_mst(I_PC)=BPC, flush else, next_mst(I_PC)=NPC+4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=RegB=dspx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_mst(I_PC)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BF/S disp</td>
<td>IPC+4</td>
<td>NPC=IPC+4</td>
<td>l/T=0/1, next_mst(I_PC)=BPC else, next_mst(I_PC)=NPC+4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT/S disp</td>
<td>IPC+4</td>
<td>NPC=IPC+4</td>
<td>l/T=0/1, next_mst(I_PC)=BPC else, next_mst(I_PC)=NPC+4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(w/ Delay slot)</td>
<td></td>
<td>RegBus=RegB=dspx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_mst(I_PC)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRA disp (w/ Delay slot)</td>
<td>IPC+4</td>
<td>NPC=IPC+4</td>
<td>next_mst(I_PC)=BPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=RegB=dspx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_mst(I_PC)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSR disp (w/ Delay slot)</td>
<td>IPC+4</td>
<td>NPC=IPC+4</td>
<td>next_mst(I_PC)=BPC</td>
<td></td>
<td>Write PR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=RegB=dspx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_mst(I_PC)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRF Rn (w/ Delay slot)</td>
<td>IPC+4</td>
<td>NPC=IPC+4</td>
<td>next_mst(I_PC)=BPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=RegB=Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_mst(I_PC)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>IPC+4</td>
<td>NPC=IPC+4</td>
<td>next$<em>{inst}$($I</em>{PC}$)=BPC</td>
<td>Write PR</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>BSRF Rn (w/ Delay slot)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=RegB=Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next$<em>{inst}$($I</em>{PC}$)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP @Rn (w/ Delay slot)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=RegB=Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next$<em>{inst}$($I</em>{PC}$)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSR Rn (w/ Delay slot)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=RegB=Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next$<em>{inst}$($I</em>{PC}$)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTS (w/ Delay slot)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=PR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next$<em>{inst}$($I</em>{PC}$)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTE (w/ Delay slot)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegBus=SPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD1=RegBus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD2=4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>next$<em>{inst}$($I</em>{PC}$)=NPC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instructions left out:
LDTLB
PREF @Rn
SLEEP
### 4.2 Possible SH3 Programming Manual Errors

<table>
<thead>
<tr>
<th>Page Num</th>
<th>Original</th>
<th>Correction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>SWAP.B</strong> Rm, Rn: Rm-&gt;Swap the bottom two bytes-&gt;REG</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td><strong>BT label</strong></td>
<td>NOT delayed branch</td>
</tr>
<tr>
<td></td>
<td><strong>BRAF operation:</strong> PC+=R[n];</td>
<td>PC+=R[n] + 4</td>
</tr>
<tr>
<td></td>
<td><strong>BSRF operation:</strong> PC+=R[n];</td>
<td>PC+=R[n] + 4</td>
</tr>
<tr>
<td></td>
<td><strong>DIV1 last paragraph:</strong> find the remainder by first finding the sum of the divisor and the quotient...</td>
<td>Product</td>
</tr>
<tr>
<td></td>
<td><strong>JSR @Rn Abstract:</strong> PC-&gt;Rn, Rn-&gt; PC</td>
<td>PC-&gt;PR</td>
</tr>
<tr>
<td></td>
<td><strong>RTE operation:</strong> PC=SPC;</td>
<td>PC=SPC+4; because RTS is shown as PC=PR+4; Both TRAPA and JSR, the two subroutine jump instructions save away PC the same way (w/o +4)</td>
</tr>
</tbody>
</table>
4.3 VHDL code

4.3.1 h_adder.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_ADDER is port (  
signal a,b: in DPWORD;
signal bzero_bar,bflip_bar,ci,en,sl,ai: in DPBIT;
signal addr_sel: in DPBITS(2 downto 0);
signal except_sel: in DPBITS(1 downto 0);
signal result, e_addr: out DPWORD;
signal z_bar,c,n,v,b31: out DPBIT
); end H_ADDER;

architecture RTL of H_ADDER is
  signal sum, except_reg: DPWORD;
begin
  process(sl, ai, a,b,bflip_bar,bzero_bar,ci)
  variable xa,xb,xsum: DPWORD;
  variable cout: DPBIT;
  begin
    -- useful for DIV1 instruction
    xa := mux2(sl,a,ai & a(31 downto 1));
    for i in 0 to DPMSB loop
      xb(i) := bzero_bar and (b(i) xor (not bflip_bar));
    end loop;
    add(xa,xb,ci,xsum,cout);
    sum <= xsum;
    z_bar <= not zero(xsum);
    n <= xsum(31);
    c <= cout;
    -- Overflow
    v <= (xa(31) and xb(31) and (not xsum(31))) or
         ((not xa(31)) and (not xb(31)) and xsum(31));
    -- Used for inst DIV0S
    b31 <= b(31);
  end process;

  -- a is muxed into e_addr primarily to take care of the inst STC.L SPC, @-Rn
  except_reg <= mux2(except_sel(0), TRA_ADDR, mux2(except_sel(1), EXPEVT_ADDR, INTEVT_ADDR));

end architecture;
e_addr <= mux2(addr_sel(2), except_reg, mux2(addr_sel(1),
mux2(addr_sel(0), a, b), sum);

-- normal result
process(sum, en)
begin
  case en is
    when '0' => result <= DPZ;
    when '1' => result <= sum;
    when others => result <= DPX;
  end case;
end process;
end RTL;
4.3.2 h_addr.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_ADDRESS is port (
    signal clk: in DPBIT;
    signal i_addr, d_addr: in DPWORD;
    signal addr_out: out DPWORD;
    signal ld_iaddr, ld_daddr, sel_oaddr: in DPBIT;
); end H_ADDRESS;

-- This unit reads the instruction and memory address and merge them into
-- one stream of address to the memory unit

architecture RTL of H_ADDRESS is
    signal saved_iaddr, saved_daddr: DPWORD;
    signal iaddr, daddr: DPWORD;
begin

    -- Save the previous address at each clock edge
    process(clk)
    begin
        if (clk'event and clk = '1') then
            saved_iaddr <= iaddr;
            saved_daddr <= daddr;
        end if;
    end process;

    -- Determine whether to use new(addr) or old (saved)
    process(i_addr, ld_iaddr)
    begin
        iaddr <= mux2(ld_iaddr, saved_iaddr, i_addr);
    end process;

    process(d_addr, ld_daddr)
    begin
        daddr <= mux2(ld_daddr, saved_daddr, d_addr);
    end process;

    -- Determine which address to output, daddr or iaddr
    process(iaddr, daddr, sel_oaddr)
    begin
        addr_out <= mux2(sel_oaddr, iaddr, daddr);
    end process;

end RTL;
4.3.3 h_boole.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

-- op(3 downto 0) determines operation performed
-- 1111  zero [used during reset]
-- 0011  select B [sethi]
-- 0111  A and B
-- 0001  A or B
-- 1001  A xor B
-- 1101  A and (not B)
-- 0100  A or (not B)
-- 0110  A xor (not B)

entity H_BOOLE is port (signal a,b: in DPWORD;
signal op: in DPBITS(3 downto 0);
signal en: in DPBIT;
signal id_a, id_constl, id_const2, id_const3, id_const4: in DPBIT;
signal scan_en: in DPBIT;
signal result: out DPWORD;
signal n,z,e: out DPBIT);
end H_BOOLE;

architecture RTL of H_BOOLE is
signal f,alt_a: DPWORD;

begin

process(ld_a)
begin
  case (ld_a) is
  when 'O' => alt_a <= DPZ;
  when '1' => alt_a <= a;
  when others => alt_a <= DPX;
  end case;
end process;

-- The constants are used for EXTS and EXTU operations

process(ld_constl)
begin
  case (ld_constl) is
  when '0' => alt_a <= DPZ;
  when '1' => alt_a <= TO_STDLOGICVECTOR(X"000000FF");
  when others => alt_a <= DPX;
  end case;
end process;

end

end H_BOOLE;
process(ld_const2)
begin
  case (ld_const2) is
  when '0' => alt_a <= DPZ;
  when '1' => alt_a <= TO_STDLOGICVECTOR(X"0000FFFF");
  when others => alt_a <= DPX;
  end case;
end process;

process(ld_const3)
begin
  case (ld_const1) is
  when '0' => alt_a <= DPZ;
  when '1' => alt_a <= TO_STDLOGICVECTOR(X"FFFFFF00");
  when others => alt_a <= DPX;
  end case;
end process;

process(ld_const4)
begin
  case (ld_const1) is
  when '0' => alt_a <= DPZ;
  when '1' => alt_a <= TO_STDLOGICVECTOR(X"FFFF0000");
  when others => alt_a <= DPX;
  end case;
end process;

process(op,alt_a,b)
variable o0,ol,o2,o3: DPBIT;
begin
  o0 := not op(0);
  ol := not op(1);
  o2 := not op(2);
  o3 := not op(3);
  for i in 0 to DPMSB loop
    f(i) <=
      mux2(b(i),mux2(alt_a(i),o0,ol),mux2(alt_a(i),o2,o3));
  end loop;
end process;

process(en,f)
begin
  case en is
  when '1' =>
    result <= f;
    n <= f(31);
    z <= zero(f);
    e <= zero(f(31 downto 24)) or zero(f(23 downto 16)) or zero(f(15 downto 8)) or zero(f(7 downto 0));
  when '0' =>
    result <= DPZ;
n <= 'X';
z <= 'X';
e <= 'X';
when others =>
  result <= DPX;
n <= 'X';
z <= 'X';
e <= 'X';
  end case;
end process;

end RTL;
library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_BYTESWAP is port (  
signal clk: in DPBIT;
signal data: in DPWORD;
signal w_data_ld: in DPBIT;
signal w_data_sel: in DPBITS(3 downto 0);
signal w_sxt_sel: in DPBITS(3 downto 0);
signal swap: out DPWORD
); end H_BYTESWAP;

-- shuffle the bytes, words, and double words in the data input from memory

architecture RTL of H_BYTESWAP is
signal w_data: DPWORD;

begin
process (clk)
begin
if (clk'event and clk = '1') then
w_data <= mux2(w_data_ld,w_data,data);
end if;
end process;

process(w_data,w_data_sel,w_sxt_sel)
variable temp,sxt: DPBITS(15 downto 0);
variable word: DPWORD;
variable sbit: DPBIT;
begin
-- Different from SPARC in that this is 32-bit input data from Mem
word := w_data;
-- Determine the sign bit
sbit := w_sxt_sel(3) and
mux2(w_sxt_sel(2),
-- byte sign extension
mux2(w_sxt_sel(1),
  mux2(w_sxt_sel(0),word(31),word(23)),
  mux2(w_sxt_sel(0),word(15),word(7)));
-- halfword sign extension
mux2(w_sxt_sel(1),word(31),word(15)))
-- Create 32 bit sxt containing all sbit
sxt := (others => sbit);
temp := mux2(w_data_sel(1),word(15 downto 0),word(31 downto
swap(31 downto 16) <= mux2(w_data_sel(3), sxt, word(31 downto 16));
swap(15 downto 8) <= mux2(w_data_sel(2), temp(15 downto 8), sxt(7 downto 0));
swap(7 downto 0) <= mux2(w_data_sel(0), temp(7 downto 0), temp(15 downto 8));
end process;
end RTL;
4.3.5 h_config.vhd

-- H_CONFIG package

library IEEE;
use IEEE.std_logic_1164.all;

package H_CONFIG is
constant DPWIDTH: natural := 32; -- width of datapath
constant INSTWIDTH: natural := 16; -- width of instruction
constant DPMSB: natural := DPWIDTH-1;
constant INSTMSB: natural := INSTWIDTH-1;
subtype DPBIT is std_logic;
subtype DPBITS is std_logic_vector;
subtype DPDWORD is std_logic_vector(DPMSB downto 0);
subtype INSTWORD is std_logic_vector(INSTMSB downto 0);
subtype DPDOUBLE is std_logic_vector(2*DPWIDTH-1 downto 0);
subtype DPBUS is std_logic_vector(DPMSB downto 0);
constant DPZERO: DPDWORD := (others => '0');
constant DPZ: DPDWORD := (others => 'Z');
constant DPX: DPDWORD := (others => 'X');
constant INSTZERO: INSTWORD := (others => '0');
constant INSTZ: INSTWORD := (others => 'Z');
constant INSTX: INSTWORD := (others => 'X');
constant N_INTERRUPTS: natural := 16;
constant INTMSB: natural := N_INTERRUPTS-1;
constant N_REG_WINDOWS: natural := 8;
constant N_REGS: natural := 16;
constant N_BANKREGS: natural := 8;
type REGISTER_FILE is array (N_REGS-1 downto 0) of DPDWORD;
type BANKREG_FILE is array (N_BANKREGS-1 downto 0) of DPDWORD;
type GLOBAL_FILE is array (15 downto 0) of DPDWORD;
constant REGMSB: natural := 3;
subtype REGADDR is DPBITS(REGMSB downto 0);
constant STCMASK: DPBITS(7 downto 0) := "00000010";
constant STCL_Mask: DPBITS(7 downto 0) := "01000011";

-- constant REG_G_PREFIX: DPBITS(4 downto 0) := "10000";
-- constant REG_G: REGADDR := REG_G_PREFIX & "0000";
-- constant REG_G_NATURAL: natural := 256;
-- constant REG_AG_MASK: REGADDR := "000001000";
constant WPTRMSB: natural := 4;
subtype WPTR is DPBITS(WPTRMSB downto 0);
constant TRA_ADDR: DPBITS(31 downto 0) :=

76
To_StdLogicVector(X"FFFFFFD0");

constant EXPEVT_ADDR: DPBITS(31 downto 0) :=
To_StdLogicVector(X"FFFFFFD4");
constant INTEVT_ADDR: DPBITS(31 downto 0) :=
To_StdLogicVector(X"FFFFFFD8");

-- CONTROL -> DP

type CTL2DP is record
  ld_inst: DPBIT;
  advance: DPBIT;
  asel,bsel: DPBITS(4 downto 0);
  isel,use_rd,rd_inc: DPBIT; -- instreg
  cwp,cwp_1: WPTR; -- instreg
  ag: DPBIT; -- instreg
  ew_data_sel,ew_data_en: DPBIT; -- wdata register
  w_data_en: DPBITS(1 downto 0); -- wdata register
  e_align: DPBITS(2 downto 0); -- wdata register
  w_data_ld: DPBIT; -- byte swap
  w_data_sel: DPBITS(4 downto 0); -- byte swap
  w_sxt_sel: DPBITS(3 downto 0); -- byte swap
  w_reg_sel: DPBIT; -- result reg
  ebldimm,ebldrslt,ebldrbb,eladrslt,ealdra: DPBIT; -- e regs
  c13_sel_bar,c22_sel_bar: DPBIT; -- e regs
  shift_sxt,shift_left_en,shift_right_en: DPBIT; -- shifter
  boole_op: DPBITS(3 downto 0); -- boole unit
  boole_en: DPBIT; -- boole unit
  scan_en: DPBIT; -- boole unit
  y_1sb_in: DPBIT; -- y reg
  y_shift_left: DPBIT; -- y reg
  y_any_shift: DPBIT; -- y reg
  y_load: DPBIT; -- y reg
  y_en: DPBIT; -- y reg
  mac_y_en_bar,mac_unsigned,mac_select: DPBIT; -- multiplier
  amsb,shift_a,adder_cin,shift_adder_out,a_en: DPBIT; -- adder
  bflip_bar,bzero_bar: DPBIT; -- adder
  addr_inc: DPBIT; -- adder
  ext_en: DPBIT; -- asr
  timer_we,timer_oe: DPBIT; -- asr
  bcount,bcounter_we,bcounter_oe: DPBIT; -- asr
  wim_we,wim_oe: DPBIT; -- asr
  wsel: WPTR; -- asr
  as1,t_ld,tbr_en,epc_en,br_sel: DPBIT; -- pc
  i64: DPBIT; -- pc
-- pragma translate_off

end record;
-- DP -> CONTROL

type DP2CTL is record
  -- pragma translate_off
      trace_pc,e_a,e_b,e_result,e_addr:DPWORD;  -- just for debugging
  -- pragma translate_on
      r_inst_bar: DPWORD;
      cp_inst: DPBIT;
      boole_z,boole_n: DPBIT;
      ylsb_next_bar: DPBIT;
      s31n,alu_z_bar,av31,bv31,alu_c: DPBIT;
      timer_zero: DPBIT;
      bcounter_zero: DPBIT;
      wbit: DPBITS(1 downto 0);
      addr_lo: DPBITS(2 downto 0);
      iaddr2: DPBIT;
end record;

-- instruction decode

type DECODE is record
  op0_sethi: DPBIT;
  op0_unconditional_branch: DPBIT;
  op0_conditional_branch: DPBIT;
  op0_branch: DPBIT;
  op0_decbne: DPBIT;
  op0_conditional_call: DPBIT;
  op0_cond: DPBITS(3 downto 0);
  op0_annul: DPBIT;
  op1_call: DPBIT;
  op2_subtract: DPBIT;
  op2_mac: DPBIT;
  op2_udivcc: DPBIT;
  op2_cpop: DPBIT;
  op2_jmpl: DPBIT;
  op2_rett: DPBIT;
  op2_restore: DPBIT;
  op2_save: DPBIT;
  op2_new_window: DPBIT;
  op2_load_cwp: DPBIT;
  op2_load_alu_cc: DPBIT;
  op2_load_boole_cc: DPBIT;
  op3_st: DPBIT;
  op3_ld: DPBIT;
  op3_lddc: DPBIT;
  op3_stdc: DPBIT;
  coproc: DPBIT;
  unimp: DPBIT;
  hah_operands: DPBIT;
  reg_dest: DPBIT;
  immediate: DPBIT;
  use Asi: DPBIT;
sxt: DPBIT;
size: DPBITS(2 downto 0);
shift_sxt: DPBIT;
shift_left: DPBIT;
shift_right: DPBIT;
boole_en: DPBIT;
scan_en: DPBIT;
boole_op: DPBITS(3 downto 0);
y_shift_left: DPBIT;
y_any_shift: DPBIT;
y_load: DPBIT;
y_en: DPBIT;
mac_y_en: DPBIT;
mac_unsigned: DPBIT;
shift_a: DPBIT;
cin: DPBITS(1 downto 0);
bflip: DPBIT;
bzero: DPBIT;
adder_en: DPBIT;
shift_adder_en: DPBIT;
tbr_en: DPBIT;
epc_en: DPBIT;
ext_en: DPBIT;
asr_reg: DPBITS(4 downto 0);
asr_oe: DPBIT;
psr_oe: DPBIT;
psr_we: DPBIT;
wim_oe: DPBIT;
wim_we: DPBIT;
tbr_ld: DPBIT;
end record;

function mux2(sel,d0,dl: DPBIT) return DPBIT;
function mux2(sel: DPBIT;d0,dl: DPBITS) return DPBITS;
function mux2(sel: DPBIT;d0,dl: integer) return integer;
-- function reg_window(cwp,cwp_1: WPTR;ag: DPBIT;rin: DPBITS(4
downto 0);
--    rd_inc: DPBIT) return REGADDR;
-- function reg_window_natural(cwp,cwp_1: WPTR;ag: DPBIT;rin:
DPBITS(4 downto 0);
--    rd_inc: DPBIT) return NATURAL;
procedure add(a,b: in DPBITS; cin: in DPBIT;
    sum: out DPBITS; cout: out DPBIT);
procedure bits_to_natural(b: in DPBITS; v,valid: out natural);
function bits_to_integer(b: in DPBITS) return INTEGER;
function bits_to_integer_sxt(b: in DPBITS) return INTEGER;
function zero(v: DPBITS) return DPBIT;
function inc(v: DPBITS) return DPBITS;
function dec(v: DPBITS) return DPBITS;
-- added functions that converts from bit(s) to
std_logic(_vector)

function To_SL (B: Bit) return Std Logic;
function To_SLV (B: Bit Vector) return Std Logic Vector;
end H_CONFIG;

package body H_CONFIG is

-- ripple increment
function inc(v: DPBITS) return DPBITS is
    variable xv,s: DPBITS(v'length-1 downto 0);
    variable c: DPBIT;
begin
    xv := v; -- make subscripts 0 based
    c := '1';
    for i in 0 to v'length-1 loop
        s(i) := xv(i) xor c;
        c := xv(i) and c;
    end loop;
    return s;
end inc;

-- ripple decrement
function dec(v: DPBITS) return DPBITS is
    variable xv,s: DPBITS(v'length-1 downto 0);
    variable c: DPBIT;
begin
    xv := v; -- make subscripts 0 based
    c := '0';
    for i in 0 to v'length-1 loop
        s(i) := xv(i) xor '1' xor c;
        c := xv(i) or c;
    end loop;
    return s;
end dec;

-- simple zero test
function zero(v: DPBITS) return DPBIT is
    variable xv: DPBITS(v'length-1 downto 0);
    variable z: DPBIT;
begin
    xv := v; -- make subscripts 0 based
    z := xv(0);
    for i in 1 to v'length-1 loop
        z := z or xv(i);
    end loop;
    return not z;
end zero;

-- convert DPBITS to a natural
procedure bits_to_natural(b: in DPBITS; v,valid: out natural) is
    variable result,okay: natural;
begin
  result := 0;
  okay := 1;
  for i in b'left downto b'right loop
    case b(i) is
      when '0' => result := result + result;
      when '1' => result := result + result + 1;
      when others => okay := 0;
    end case;
  end loop;
  v := result;
  valid := okay;
end bits_to_natural;

-- convert DPBITS to an integer
function bits_to_integer(b: in DPBITS) return INTEGER is
  variable result: integer;
begin
  result := 0;
  for i in b'left downto b'right loop
    if (b(i) = '1') then result := result + result + 1;
    else result := result + result;
  end if;
  end loop;
  return result;
end bits_to_integer;

-- convert DPBITS to a sign-extended integer
function bits_to_integer_sxt(b: in DPBITS) return INTEGER is
  variable result: integer;
begin
  if (b(b'left) = '1') then result := -1;
  else result := 0;
  end if;
  for i in b'left downto b'right loop
    if (b(i) = '1') then result := result + result + 1;
    else result := result + result;
  end if;
  end loop;
  return result;
end bits_to_integer_sxt;

-- simple ripple adder is useful for simulation
procedure add(a,b: in DPBITS; cin: in DPBIT;
  sum: out DPBITS; cout: out DPBIT) is
variable c: DPBIT;
variable xa, s: DPBITS(a'length-1 downto 0);
variable xb: DPBITS(b'length-1 downto 0);
begin
xa := a; -- make subscripts 0 based
xb := b; -- make subscripts 0 based
c := cin;
for i in 0 to a'length-1 loop
s(i) := xa(i) xor xb(i) xor c;
c := (xa(i) and xb(i)) or (xa(i) and c) or (xb(i) and c);
end loop;
sum := s;
cout := c;
end add;

-- translate logical register number to physical register file address.
-- function reg_window(cwp,cwp_1: WPTR;ag: DPBIT;rin: DPBITS(4 downto 0); rd_inc: DPBIT) return REGADDR is
-- variable result: REGADDR;
-- begin
-- case rin(4 downto 3) is
-- when "00" => result := REG_G_PREFIX & ag & rin(2 downto 0);
-- when "01" => result := "0" & cwp_1(REGMSB-5 downto 0) & rin(3 downto 0);
-- when "10" | "11" => result := "0" & cwp(REGMSB-5 downto 0) & rin(3 downto 0);
-- when others => result := (others => 'X');
-- end case;
-- result(0) := result(0) or rd_inc;
-- return result;
-- end;

-- function reg_window_natural(cwp,cwp_1: WPTR;ag: DPBIT;rin: DPBITS(4 downto 0); rd_inc: DPBIT) return NATURAL is
-- variable result: REGADDR;
-- variable addr,valid: natural;
-- begin
-- result := reg_window(cwp,cwp_1,ag,rin,rd_inc);
-- bits_to_natural(result,addr,valid);
-- if (valid = 0) then addr := REG_G0_NATURAL; end if;
-- return addr;
-- end;

function mux2(sel,d0,d1: DPBIT) return DPBIT is
variable result: DPBIT;
begin
    case sel is
when '1' => result := d1;
when '0' => result := d0;
when others => if (d0 = d1) then result := d0; else result := 'X'; end if;
end case;
return result;
end;

function mux2(sel: DPBIT; d0, d1: DPBITS) return DPBITS is
  variable a, b, result: DPBITS(d0'length-1 downto 0);
begin
  a := d0; -- resolve indexing problems
  b := d1;
  case sel is
    when '1' => result := d1;
    when '0' => result := d0;
    when others =>
      for i in 0 to d0'length-1 loop
        if (a(i) = b(i)) then result(i) := a(i);
        else result(i) := 'X';
        end if;
      end loop;
  end case;
  return result;
end;

function mux2(sel: DPBIT; d0, d1: integer) return integer is
  variable result: integer;
begin
  case sel is
    when '1' => result := d1;
    when '0' => result := d0;
    when others => if (d0 = d1) then result := d0; else result := 0; end if;
  end case;
  return result;
end;

-- added functions that converts from bit(s) to std_logic(_vector)

function To_SL (B: Bit) return Std_Logic is
begin
  case B is
    when '0' => return '0';
    when '1' => return '1';
  end case;
end To_SL;

function To_SLV (B: Bit_Vector) return Std_Logic_Vector is
variable result: Std_Logic_Vector (B'range);
begin
For i in B'range loop
  case B(i) is
    when '0' => result(i) := '0';
    when '1' => result(i) := '1';
  end case;
end loop;

return result;
end To_SLV;
end H_CONFIG;
library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_EREGS is port (  
signal clk: in DPBIT;  
signal rega, regb, regc: in DPWORD;  
signal e_result: in DPWORD;  
signal advance: in DPBIT;  
signal ealdrslt, ebldrslt: in DPBIT;  
signal ealdra, ealdr, ealdipc: in DPBIT;  
signal ealdone, ealdtwo, ealdfour: in DPBIT;  
signal ebldrdb, eblrdrc: in DPBIT;  
-- Useful for shift instructions  
signal ebldone, ebldtwo, ebldfour, ebldeight, ebldsixteen: in DPBIT;  
  
signal e_a,e_b: out DPWORD  
); end H_EREGS;

-- Pipelines data onto ea, eb  
-- regc is muxed into both ea and eb, ec nolonger needed  
-- All regs have feedback for advancement control

architecture RTL of H_EREGS is  
signal local_a,local_b: DPWORD;  
signal eadat,ebdat: DPWORD;  
begin

process(local_a,advance)  
begin  
case (advance) is  
when '1' => eadat <= DPZ;  
when '0' => eadat <= local_a;  
when others => eadat <= DPX;  
end case;  
end process;  

process(rega,ealdra)  
begin  
case (ealdra) is  
when '0' => eadat <= DPZ;  
when '1' => eadat <= rega;  
when others => eadat <= DPX;  
end case;  
end process;  

process(regc, ealdrc)
begin
  case (ealdrc) is
    when '0' => eadat <= DPZ;
    when '1' => eadat <= regc;
    when others => eadat <= DPX;
  end case;
end process;

-- e_result is forwarded here because it doesn't become available
-- late in the E stage. Therefore it is sent to the later of the
-- in R stage

process(e_result, ealdrslt)
begin
  case (ealdrslt) is
    when '0' => eadat <= DPZ;
    when '1' => eadat <= e_result;
    when others => eadat <= DPX;
  end case;
end process;

process(ealdone)
begin
  case (ealdone) is
    when '0' => eadat <= DPZ;
    when '1' => eadat <= To_StdLogicVector(X"1");
    when others => eadat <= DPX;
  end case;
end process;

process(ealdtwo)
begin
  case (ealdtwo) is
    when '0' => eadat <= DPZ;
    when '1' => eadat <= TO_STDLOGICVECTOR(X"2");
    when others => eadat <= DPX;
  end case;
end process;

process(ealdfour)
begin
  case (ealdfour) is
    when '0' => eadat <= DPZ;
    when '1' => eadat <= TO_STDLOGICVECTOR(X"4");
    when others => eadat <= DPX;
  end case;
end process;

86
process(local_b,advance)
begin
  case (advance) is
    when '1' => ebdat <= DPZ;
    when '0' => ebdat <= local_b;
    when others => ebdat <= DPX;
  end case;
end process;

process(regb,ebldrb)
begin
  case (ebldrb) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= regb;
    when others => ebdat <= DPX;
  end case;
end process;

process(regc,ebldrc)
begin
  case (ebldrc) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= regc;
    when others => ebdat <= DPX;
  end case;
end process;

process(e_result,ebldrslt)
begin
  case (ebldrslt) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= e_result;
    when others => ebdat <= DPX;
  end case;
end process;

process(ebldone)
begin
  case (ebldone) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= To_StdLogicVector(X"1");
    when others => ebdat <= DPX;
  end case;
end process;

process(ebldtwo)
begin
  case (ebldtwo) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= TO_STDLOGICVECTOR(X"2");
    when others => ebdat <= DPX;
  end case;
end process;
end case;
end process;

process(ebldfour)
begin
  case (ebldfour) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= TO_STDLOGICVECTOR(X"4");
    when others => ebdat <= DPX;
  end case;
end process;

process(Debeight)
begin
  case (ebldfour) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= TO_STDLOGICVECTOR(X"8");
    when others => ebdat <= DPX;
  end case;
end process;

process(ebldsixteen)
begin
  case (ebldfour) is
    when '0' => ebdat <= DPZ;
    when '1' => ebdat <= TO_STDLOGICVECTOR(X"10");
    when others => ebdat <= DPX;
  end case;
end process;

-- Not necessary any more since imm/disp as well as gbr data is driven onto regb in result reg
--
-- process(imm, ebldimm)
-- begin
--   case (ebldimm) is
--     when '0' => ebdat <= DPZ;
--     when '1' => ebdat <= imm;
--     when others => ebdat <= DPX;
--   end case;
-- end process;
--
--
-- process(gbr, ebldgbr)
-- begin
--   case (ebldgbr) is
--     when '0' => ebdat <= DPZ;
--     when '1' => ebdat <= gbr;
--     when others => ebdat <= DPX;
--   end case;
end process;

process(clk)
begin
  if (clk'event and clk = '1') then
    local_a <= eadat;
    local_b <= ebdat;

  end if;
  e_a <= local_a;
  e_b <= local_b;
end process;
end RTL;
library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_INSTREG is port (  
signal clk: in DPBIT;
signal d: in DPWORD;
signal id_inst, inst_sel: in DPBIT;
-- q is internal (to DP module), q_bar is going to control logic
signal q,q_bar: out INSTWORD;
signal rinst11_bar,rinst7_bar: out DPBIT;
signal rs0,rsl: out REGADDR;
signal rs0type: out DPBIT -- Indicates whether rs0 refers to a banked reg
); end H_INSTREG;

-- Accepts data output from memory and extracts the 16 bit instruction
-- to be outputed to both other modules w/i the DP as well as control
-- module to generate control signals. Outputs register select 0 and 1,
-- outputs r_inst as q and q_bar for both DP and control logic
-- outputs sign bit for 8-bit and 12-bit disp/imm data

architecture RTL of H_INSTREG is
signal inst_reg: DPWORD; -- saves incoming d data
signal r_inst: INSTWORD;
begin
-- latch incoming instruction data, which is 32 bits
process(clk)
begin
if (clk'event and clk = '1') then
   inst_reg <= mux2(ld_inst,inst_reg,d);
end if;
end process;

-- Break the 32 bit word to two 16 bit instructions and outputs the
-- info relevant to the current instruction.
process(inst_reg, inst_sel)
begin
   -- select half of the 32 bits
   r_inst <= mux2(inst_sel, inst_reg(15 downto 0), inst_reg(31 downto 16));
   q <= inst;
   q_bar <= not inst;
   rinst11_bar <= not inst(11);
   rinst7_bar <= not inst(7);
end process;

-- Some decoding to determines register select 0 and 1 and the
-- type of
-- register rs0 refers to. rs0type
-- Corresponds to rd0type in REGFIL. Bank info provided to
REGFIL by RB
-- bit from SR
process(r_inst)
begin
  rs0 <= r_inst(7 downto 4);
  rsl <= r_inst(11 downto 8);
-- rs0type = 0 if rs0 is not a banked register.  1 otherwise
-- Determine whether the instruction is either STC Rx_bank or
STC.L Rx_bank
  if( zero(r_inst(15 downto 12) & r_inst(3 downto 0) and not
STC_MASK) = 1
    or zero(r_inst(15 downto 12) & r_inst(3 downto 0) and not
STCL_MASK) = 1) then
    rs0type <= 1;
  else
    rs0type <= 0;
  end if;
end process;

end RTL;
4.3.7 h_pc.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_PC is port (
    signal clk: in DPBIT;
    -- connected to w_pc_result from result_reg
    signal w_spr_result: in DPWORD;
    signal regb: inout DPWORD;
    signal e_addr: in DPWORD;
    signal iaddr: out DPWORD;
    signal iaddr2: out DPBIT;
    signal daddr: out DPWORD;
    signal daddr_lo: out DPBITS(1 downto 0);

    signal c_x100_ctl, npcctl, rpcctl, c_4_ctl: in DPBIT;
    signal advance, gbr_ld, vbr_ld, ssr_ld, spc_ld, pr_ld: in DPBIT;
    signal pc_spc, pc_pr: in DPBIT;
    signal br_2_ipc, npc_2_ipc, pwron: in DPBIT;
    signal npc_sel: in DPBIT;
    signal ld_rpc, ld_gbr, ld_vbr, ld_ssr, ld_sp, ld_pc, ld_regb: DPBIT;
    signal ld_epc, ld_cpc, ld_wpc: DPBIT;
    signal ld_reg_bus: DPBIT
); end H_PC;

architecture RTL of H_PC is
    signal vbr, pr, spc, ssr, gbr: DPWORD;
    signal npc, bpc, inc_pc, alt_bpc, i_pc, r_pc, e_pc, c_pc, w_pc: DPWORD;
    signal addendl, addend2, local_addr: DPWORD;
    signal reg_bus: DPWORD;

begin
    -- internal registers

    -- ADDEND1
    addendl <= reg_bus;

    -- ADDEND2
    process(c_x100_ctl)
    begin
        case (c_x100_ctl) is
        when '0' => addend2 <= DPZ;
        when '1' => addend2 <= To_SLV(X"00000100");
        when others => addend2 <= DPX;
        end case;
    end process;
-- usually that’s value of pc+4 (for most branches, pc = pc+4 + X, etc)
process(npc, npcctl)
begin
  case (npcctl) is
    when '0' => addend2 <= DPZ;
    when '1' => addend2 <= npc;
    when others => addend2 <= DPX;
  end case;
end process;

-- Usually that’s the value of pc (for BSRF, pc = rn + pc ,etc)
-- process(r_pc, rpcctl)
begin
  case (rpcctl) is
    when '0' => addend2 <= DPZ;
    when '1' => addend2 <= r_pc;
    when others => addend2 <= DPX;
  end case;
end process;

-- supplies constant 4
process(c_4_ctl)
begin
  case (c_4_ctl) is
    when '0' => addend2 <= DPZ;
    when '1' => addend2 <= To_SLV(X"00000004");
    when others => addend2 <= DPX;
  end case;
end process;

-- used by branches, calls. Perform addition
process(addendl, addend2)
variable s: DPWORD;
variable ignore: DPBIT;
begin
  add(addendl, addend2,'0',s,ignore);
bpc <= s;
end process;

-- increment i_pc by 4. This probably belong outside of clk
process
process(i_pc)
begin
  inc_pc <= inc(i_pc(DPMSB downto 2)) & "00";
end process;

process(clk)
begin
-- latches
if (clk'event and clk = '1') then

alt_bpc <= mux2(advance, alt_bpc, bpc);
npc <= mux2(advance, mux2(npc_sel, i_pc, npc), inc_pc);
r_pc <= mux2(advance, r_pc, i_pc);
e_pc <= mux2(advance, e_pc, r_pc);
c_pc <= mux2(advance, c_pc, e_pc);
w_pc <= mux2(advance, w_pc, c_pc);

gbr <= mux2(gbr(ld, gbr, w_spr_result);
vbr <= mux2(vbr(ld, vbr, w_spr_result);
ssr <= mux2(ssr(ld, ssr, w_spr_result);
spr <= mux2(spr(ld, spr, mux2(pc_spr, w_spr_result, w_pc)));
pr <= mux2(pr(ld, pr, mux2(pc_pr, w_spr_result, w_pc)));
local_addr <= mux2(advance, local_addr, e_addr);

end if;
end process;

process(br_2_ipc, bpc)
begin
  case (brsel) is
    when '0' => i_pc <= DPZ;
    when '1' => i_pc <= alt_bpc;
    when others => i_pc <= DPX;
  end case;
end process;

process(npc_2_ipc, npc)
begin
  case (npcsel) is
    when '0' => i_pc <= DPZ;
    when '1' => i_pc <= npc;
    when others => i_pc <= DPX;
  end case;
end process;

process(pwron)
begin
  case (pwron) is
    when '0' => i_pc <= DPZ;
    when '1' => i_pc <= To_SLV(X"10000000");
    when others => i_pc <= DPX;
  end case;
end process;

process(ld_rpc, r_pc)
begin

case (ld_rpc) is
  when '0' => reg_bus <= DPZ;
  when '1' => reg_bus <= r_pc;
  when others => reg_bus <= DPX;
end case;
end process;

process (ld_gbr, gbr)
begin
  case (ld_gbr) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= gbr;
    when others => reg_bus <= DPX;
  end case;
end process;

process (ld_vbr, vbr)
begin
  case (ld_vbr) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= vbr;
    when others => reg_bus <= DPX;
  end case;
end process;

process (ld_ssr, ssr)
begin
  case (ld_ssr) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= ssr;
    when others => reg_bus <= DPX;
  end case;
end process;

process (ld_spc, spc)
begin
  case (ld_spc) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= spc;
    when others => reg_bus <= DPX;
  end case;
end process;

process (ld_pr, pr)
begin
  case (ld_pr) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= pr;
    when others => reg_bus <= DPX;
  end case;
end process;
-- ld_epc, ld_cpc and ld_wpc bypass E_, C_ and W_ stage PC value back to R stage
-- This is data source for Special Registers. The bypassing is just like
-- that done for the RF

process(ld_epc, e_pc)
begin
  case (ld_epc) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= e_pc;
    when others => reg_bus <= DPX;
  end case;
end process;

process(ld_cpc, c_pc)
begin
  case (ld_cpc) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= c_pc;
    when others => reg_bus <= DPX;
  end case;
end process;

process(ld_wpc, w_pc)
begin
  case (ld_wpc) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= w_pc;
    when others => reg_bus <= DPX;
  end case;
end process;

process(ld_regb, regb)
begin
  case (ld_regb) is
    when '0' => reg_bus <= DPZ;
    when '1' => reg_bus <= regb;
    when others => reg_bus <= DPX;
  end case;
end process;

process(reg_bus, ld_reg_bus)
begin
  case(ld_reg_bus) is
    when '0' => regb <= DPZ;
    when '1' => regb <= reg_bus;
    when others => regb <= DPX;
  end case;
end process;
iaddr <= i_pc;
iaddr2 <= i_pc(2);
daddr <= local_addr;
daddr_lo <= local_addr(1 downto 0);

end RTL;
library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_REGDRIVE is port
(
  signal rd0, rdl, r0: in DPWORD;
  signal SR: in DPWORD; -- input from control logic
  signal a_en_rd0, a_en_rdl, zero_ra_bar: in DPBIT;
  signal b_en_rd0, b_en_rdl, b_en_constl, b_en_SR: in DPBIT;
  signal c_en_r0: in DPBIT;
  signal rega, regb, regc: out DPWORD
); end H_REGDRIVE;

-- Drives data output from RF onto rega, regb, and regc, also
-- drives zero to rega
-- rd0 and rdl will be driven to both RegA and RegB

architecture RTL of H_REGDRIVE is
begin
  process(a_en_rd0, rd0)
  begin
    case a_en_rd0 is
      when '1' => rega <= rd0;
      when '0' => rega <= DPZ;
      when others => rega <= DPX;
    end case;
  end process;

  process(a_en_rdl, rdl)
  begin
    case a_en_rdl is
      when '1' => rega <= rdl;
      when '0' => rega <= DPZ;
      when others => rega <= DPX;
    end case;
  end process;

  process(zero_ra_bar)
  begin
    case zero_ra_bar is
      when '0' => rega <= DPZERO;
      when '1' => rega <= DPZ;
      when others => rega <= DPX;
    end case;
  end process;

  process(b_en_rd0, rd0)
  begin

case b_en_rd0 is
  when '1' => regb <= rd0;
  when '0' => regb <= DPZ;
  when others => regb <= DPX;
end case;
end process;

process(b_en_rd1,rd1)
begin
  case b_en_rd1 is
    when '1' => regb <= rd1;
    when '0' => regb <= DPZ;
    when others => regb <= DPX;
  end case;
end process;

-- This is for TRAPA instruction where 0x160 is loaded into EXPEVT register
process(b_en_const1)
begin
  case b_en_const1 is
    when '1' => regb <= To_StdLogicVector(X"160");
    when '0' => regb <= DPZ;
    when others => regb <= DPX;
  end case;
end process;

-- This is for loading SR from control logic to RegB, likely to need mod
process(b_en_sr)
begin
  case b_en_sr is
    when '1' => regb <= SR;
    when '0' => regb <= DPZ;
    when others => regb <= DPX;
  end case;
end process;

process(c_en_r0,r0)
begin
  case c_en_r0 is
    when '1' => regc <= r0;
    when '0' => regc <= DPZ;
    when others => regc <= DPX;
  end case;
end process;
end process;
end RTL;
4.3.9 h_regfil.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_REGFILE is port (  
signal clk: in DPBIT;
  signal rdadd0, rdadd1, wradd0, wradd1: in REGADDR;
-- type indicates whether reg address refers to a banked reg.
-- whether it's banked (use LSB 3 bits) or non-banked (4 bits)
  signal rd0type, wr1en, wr1type: in DPBIT;
-- bank selects whether to read bank 0 or bank 1, from RB bit of SR
  signal rd0bank, wrlbank: in DPBIT;
  signal wdin0, wdin1: in DPWORD;
  signal rd0out, rdlout, r0out: out DPWORD
); end H_REGFILE;

architecture RTL of H_REGFILE is
  signal regfile: REGISTER_FILE;
  signal bankreg: BANKREG_FILE;
begin
  -- read port 0, active during clk low
  process(clk, rdadd0, rd0type, rd0bank)
    variable valid, addr: natural;
  begin
    -- Checks to see if Rd0 refers to banked
    if (rd0type = '0') then
      bits_to_natural(rdadd0(REGMSB downto 0), addr, valid);
    else
      bits_to_natural(rdadd0(REGMSB-1 downto 0), addr, valid);
    end if;
    case clk is
    when '1' => rd0out <= DPX;
    when '0' =>
      if (valid = 1) then
        -- Select bank to use
        if (rd0bank = '0') then
          rd0out <= regfile(addr);
        else
          rd0out <= bankreg(addr);
        end if;
      else rd0out <= DPX;
    end if;
    when others => rd0out <= DPX;
  end case;
  end process;

  -- read port 1, active during clk low
  process(clk, rdadd1)
variable valid, addr: natural;
begin
  bits_to_natural(rdaddl(REGMSB downto 0), addr, valid);

  case clk is
    when '1' => rd1out <= DPX;
    when '0' =>
      if (valid = 1) then
        rd1out <= regfile(addr);
      else rd1out <= DPX;
      end if;
    when others => rd1out <= DPX;
  end case;
end process;

-- read register 0, active during clk low
process(clk)
begin
  case clk is
    when '1' => r0out <= DPX;
    when '0' => r0out <= regfile(0);
    when others => r0out <= DPX;
  end case;
end process;

-- write port 0, writes happen during clk low
process(clk, wdin0, wradd0)
variable valid, addr: natural;
begin
  bits_to_natural(wradd0(REGMSB downto 0), addr, valid);

  case clk is
    when '1' => null;
    when '0' =>
      if (valid = 1) then
        regfile(addr) <= wdin;
      else
        for i in 0 to regfile'length-1 loop
          regfile(i) <= DPX;
        end loop;
      end if;
    when others =>
      for i in 0 to regfile'length-1 loop
        regfile(i) <= DPX;
      end loop;
  end case;
end process;
-- write port 1, writes happen during clk low
-- wrlen: enables the usage of the second write port
-- wrltype: distinguishes banked register address from gpr address
-- wrlbank: indicates which bank if writing to banked registers

process(clk, wdinl, wraddl, wrlen, wrltype, wrlbank)
variable valid, addr: natural;
begin
  if(wrlen = '1') then
    if(wrltype = '0') then
      bits_to_natural(wraddl(REGMSB downto 0), addr, valid);
    elsif(wrltype = '1') then
      bits_to_natural(wraddl(REGMSB-1 downto 0), addr, valid);
    end if;
  end if;

  case clk is
  when '1' => null;
  when '0' =>
    if (valid = 1) then
      if(wrlbank = '0') then
        regfile(addr) <= wdin;
      else
        bankreg(addr) <= wdin;
      end if;
    else
      for i in 0 to bankreg'length-1 loop
        bankreg(i) <= DPX;
      end loop;
    end if;
  when others =>
    for i in 0 to bankreg'length-1 loop
      bankreg(i) <= DPX;
    end loop;
  end case;

end if;

end process;

end RTL;
4.3.10  h_rsltrg.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_RESULTREG is port (  
signal clk: in DPBIT;
signal advance: in DPBIT;
signal ea, eb, e_result: in DFPWORD;
signal mdata: in DFPWORD;
signal swp_data: in DFPWORD;
signal r_inst: in INSTWORD;
signal swap, swap_word: in DPBIT;
signal eldea, eldeb, eldxtrct, elderslt: in DPBIT;

signal sxt: in DPBIT; -- For imm/disp, contains the extension bit
signal shift: in DPBITS(1 downto 0); -- For imm/disp, shifts imm/disp by 0, 1, or 2
signal sel: in DPBITS(5 downto 0); -- For imm/disp, selects the position of extension
signal ra_ld_imm, rb_ld_imm, rc_ld_imm: in DPBIT;

signal cr_ra_bypass, cr_rb_bypass: in DPBIT;
signal wr_ra_bypass, wr_rb_bypass: in DPBIT;
signal sr_ra_bypass, sr_rb_bypass: in DPBIT; -- bypass using swp_data

signal w_result_sel: in DPBIT;
signal rega, regb, regc, w_result0, w_result1: out DFPWORD
);
end H_RESULTREG;

-- Provides register write data for both GPR and SPR (Special Purpose Registers)
-- Bypasses various data onto bus rega and regb. They both get pipelined
-- c_result and w_result (from e_result), as well swap data which is mem
-- data output pipelined into w stage. In addition, regb gets imm/disp
-- extracted from r_inst. Another output is wresult which is e_result
-- pipelined into w stage.

architecture RTL of H_RESULTREG is
signal c_result, w_result: DFPWORD;
signal imm_disp: DFPWORD;
signal eb_alt, xtrct, edat: DFPWORD;
begin
process(ea, eb, swap, swap_word)
    variable eb_swap_byte, eb_swap_word: DPWORD;
begin
    -- Recombine ea and eb to satisfy instructions SWAP and XTRCT
    eb_swap_byte := ea(31 downto 16) & ea(7 downto 0) & eb(15 downto 8); -- swap.b
    eb_swap_word := eb(15 downto 0) & eb(31 downto 16); -- swap.w
    -- swap determines whether the instruction is swap. swap_word determines whether
    -- it's swap.b or swap.w
    eb_alt <= mux2(swap, ea, mux2(swap_word, eb_swap_byte, eb_swap_word));
    -- For XTRCT
    xtrct <= ea(31 downto 16) & eb(15 downto 0);
end process;

-- Drive edat
process(eldea, eldeb, eldxtrct, elderslt, ea, eb_alt, xtrct, e_result)
begin
    case (eldea) is
        when '1' => edat <= ea;
        when '0' => edat <= DPZ;
        when others => edat <= DPX;
    end case;
    case (eldeb) is
        when '1' => edat <= eb_alt;
        when '0' => edat <= DPZ;
        when others => edat <= DPX;
    end case;
    case (eldxtrct) is
        when '1' => edat <= xtrct;
        when '0' => edat <= DPZ;
        when others => edat <= DPX;
    end case;
    case (elderslt) is
        when '1' => edat <= e_result;
        when '0' => edat <= DPZ;
        when others => edat <= DPX;
    end case;
end process;

-- Pipelines register write data
process(clk)
    if (clk'event and clk = '1') then
        c_result <= mux2(advance, c_result, edat);
        w_result <= mux2(advance, w_result, c_result);
    end if;
end process;

-- Bypasses c_result to rega and regb
process(c_result, cr_ra_bypass, cr_rb_bypass)
begin
  case cr_ra_bypass is
    when '1' => rega <= c_result;
    when '0' => rega <= DPZ;
    when others => rega <= DPX;
  end case;
  case cr_rb_bypass is
    when '1' => regb <= c_result;
    when '0' => regb <= DPZ;
    when others => regb <= DPX;
  end case;
end process;

-- w_result and swp_data (memory out), both in W stage,
-- are each bypassed onto rega and regb separately. This is to facilitate
-- simultaneous write to general purpose register file and special registers

process(w_result, wr_ra_bypass, wr_rb_bypass)
begin
  case wr_ra_bypass is
    when '1' => rega <= w_result;
    when '0' => rega <= DPZ;
    when others => rega <= DPX;
  end case;
  case wr_rb_bypass is
    when '1' => regb <= w_result;
    when '0' => regb <= DPZ;
    when others => regb <= DPX;
  end case;
end process;

-- bypass swap data into ra and rb
process(swp_data, sr_ra_bypass, sr_rb_bypass)
begin
  case sr_ra_bypass is
    when '1' => rega <= swp_data;
    when '0' => rega <= DPZ;
    when others => rega <= DPX;
  end case;
  case sr_rb_bypass is
    when '1' => regb <= swp_data;
    when '0' => regb <= DPZ;
    when others => regb <= DPX;
  end case;
end process;
-- bypass mdata into ra and rb, useful in atomic operations such as AND
-- .b #imm, @(R0, GBR)
-- case md_ra_bypass is
--   when '1' => rega <= mdata;
--   when '0' => rega <= DPZ;
--   when others => rega <= DPX;
--   end case;
-- case md_rb_bypass is
--   when '1' => regb <= mdata;
--   when '0' => regb <= DPZ;
--   when others => regb <= DPX;
--   end case;

-- Bypasses imm/disp data to rega, regb, and regc
process(ra_ld_imm, rb_ld_imm, rc_ld_imm, imm_disp)
begin
  case ra_ld_imm is
    when '1' => rega <= imm_disp;
    when '0' => rega <= DPZ;
    when others => rega <= DPX;
  end case;

  case rb_ld_imm is
    when '1' => regb <= imm_disp;
    when '0' => regb <= DPZ;
    when others => regb <= DPX;
  end case;

  case rc_ld_imm is
    when '1' => regc <= imm_disp;
    when '0' => regc <= DPZ;
    when others => regc <= DPX;
  end case;
end process;

-- Output result to two buses in case of two simultaneous register writes
process(w_result_sel, w_result, swp_data)
begin
  w_result0 <= mux2(w_result_sel, w_result, swp_data);
  w_result1 <= mux2(w_result_sel, swp_data, w_result);
end process;

-- Extracting immediate and displacement data from r_inst.
Control signal sxt
-- provides info as to whether to extend with 1 or 0.
-- sel(5 downto 0) selects
-- data for the middle ten bits of rslt.
process(r_inst, sxt)
variable sft_inst: DPWORD;
variable rslt: DPWORD;
variable ext: DPWORD;
begin

ext := (others => sxt); -- expands to 32 bit extension

-- shift instruction for 0, 1, or 2 bits
sft_inst := ext(31 downto 16) & mux2(shift(0), r_inst,
mux2(shift(1), r_inst(15 downto 1)&’0’, r_inst(15 downto 2)&"00");)

-- selects the middle ten bits for various positions of original length
-- 4, 8, 12 into multiplying by 1, 2, or 4
rslt(31 downto 14) := ext(31 downto 14);
rslt(3 downto 0) := sft_inst(3 downto 0);
rslt(13 downto 10) := mux2(sel(0), ext(13 downto 10), r_inst(13 downto 10));
rslt(9) := mux2(sel(1), ext(9), r_inst(9));
rslt(8) := mux2(sel(2), ext(8), r_inst(8));
rslt(7 downto 6) := mux2(sel(3), ext(7 downto 6), r_inst(7 downto 6));
rslt(5) := mux2(sel(4), ext(5), r_inst(5));
rslt(4) := mux2(sel(5), ext(4), r_inst(4));

imm_disp <= rslt;
end process;

end RTL;
library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

entity H_SHIFT is port (
signal a: in DPWORD; -- corresponds to e_b, or Rn
signal b: in DPBITS(4 downto 0); -- corresponds to e_a, or Rm
signal le: in DPBIT;
signal re: in DPBIT;

-- shift in bit. It's either t, 0, a(31), or a(0)
signal sin: in DPBIT;
signal sout: out DPBIT;
signal result: out DPWORD
);

architecture RTL of H_SHIFT is
begin

process(le,a,b)
variable 11,12,14,18,116, sin_mask: DPWORD;
begin
  sin_mask := (others => sin);
  116 := mux2(b(4),a,a(15 downto 0) & sin_mask(15 downto 0));
  11 := mux2(b(0),116,116(30 downto 0) & sin);
  18 := mux2(b(3),11,11(23 downto 0) & sin_mask(7 downto 0));
  12 := mux2(b(1),18,18(29 downto 0) & sin_mask(1 downto 0));
  14 := mux2(b(2),12,12(27 downto 0) & sin_mask(3 downto 0));
  case le is
    when '1' => result <= 14;
    sout <= a(31);
    when '0' => result <= DPZ;
    sout <= 'Z';
    when others => result <= DPX;
    sout <= 'X';
  end case;
end process;

process(re,a,b)
variable r1,r2,r4,r8,r16, sin_mask: DPWORD;
begin
  sin_mask := (others => sin);
  r16 := mux2(b(4),a,sin_mask(15 downto 0) & a(31 downto 16));
  r1 := mux2(b(0),r16, sin & r16(31 downto 1));
  r8 := mux2(b(3),r1,sin_mask(7 downto 0) & r1(31 downto 8));
  r2 := mux2(b(1),r8,sin_mask(1 downto 0) & r8(31 downto 2));
  r4 := mux2(b(2),r2,sin_mask(3 downto 0) & r2(31 downto 4));
case re is
  when '1' => result <= r4;
    sout <= a(0);
  when '0' => result <= DPZ;
    sout <= 'Z';
  when others => result <= DPX;
    sout <= 'X';
  end case;
end process;
end RTL;
library IEEE;
use IEEE.std_logic_1164.all;
use work.h_config.all;

data

entity H_WDATAREG is port (  signal clk: in DPBIT;
  signal advance: in DPBIT;
  signal regb, regc, e_result: in DPWORD;
  signal rwdata_sel: in DPBITS(1 downto 0);
  signal data_en: in DPBIT;
  signal tas_b: in DPBIT; -- tas_b sets bit 7
  signal e_align: in DPBITS(1 downto 0);
  signal data: out DPWORD
  ); end H_WDATAREG;

architecture RTL of H_WDATAREG is
  signal r_wdata, e_wdata, c_wdata: DPWORD;
begin
  process (clk)
  variable result: DPWORD;
  begin

  -- pipelines data from R stage through E to C to be written to memory.
  -- Most data are from GPR or other banked and special registers.
  -- The special case is with Memory Logic Operations Instructions such as:
  -- AND.B #imm, @(RO,GBR), where (RO + GBR)&imm -> (RO + GBR)
  -- E_result bus is used both to forward data and to send ALU result to be
  -- written to memory.
  -- regc muxed in for inst such as MOV._ RO, @(disp, GBR)

  -- r_wdata sources either regb, regc, or e_result
  r_wdata <= mux2(rwdata_sel(0), e_result, mux2(rwdata_sel(1),
  regc, regb));

  if (tas_b = '1') then -- tas.b always sets bit 7 of mdata
    e_wdata(7) <= '1';
  end if;

  -- Realign data for output by duplicating 16bit and 8bit data
  if (e_align = 00) then => 32 bit data
    result(7 downto 0) := e_wdata(7 downto 0);
  elsif (e_align = 10) then => 16 bit data
    result(15 downto 8) :=
  elsif (e_align = 11) then => 8 bit data
    result(15 downto 0) := e_wdata(7 downto 0);
  end if;

end architecture;

mux2(e_align(0), e_wdata(15 downto 8), e_wdata(7 downto 0));
result(23 downto 16) :=
mux2(e_align(1), e_wdata(23 downto 16), e_wdata(7 downto 0));
result(31 downto 24) :=
mux2(e_align(1), e_wdata(31 downto 24), result(15 downto 8));

if (clk'event and clk = '1') then
  e_wdata <= mux2(advance, e_wdata, r_wdata);
  c_wdata <= mux2(advance, c_wdata, result);
end if;
end process;

-- Output data
process(c_wdata, data_en)
begin
  case data_en is
    when '1' => data <= c_wdata;
    when '0' => data <= DPZ;
    when others => data <= DPX;
  end case;
end process;
end RTL;
References


11. *CPU Info Center*: http://infopad.eecs.berkeley.edu/CIC/


