Interactive Assembly Language Editor for the Parallel Processor of the TMS320C8x

by

Mark R. Sadowski

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

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

© Mark R. Sadowski, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 24, 1996

Certified by ....
Srinivas Devadas
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUN 11 1996

Interactive Assembly Language Editor for the Parallel Processor of the TMS320C8x

by

Mark R. Sadowski

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Electrical Engineering and Computer Science

Abstract

With multiple data paths, a 3-input ALU, and the capacity for up to 4 parallel instructions per clock cycle (the equivalent of up to 10 RISC-like instructions), building an efficient compiler for the Parallel Processor (PP) of the Texas Instruments TMS320C8x family of digital signal processors (DSPs) is very difficult. Since efficiency is critical in DSP applications, assembly language is recommended to program the PP. Assembly language programming is also made very difficult, however, because of the numerous interactions between different parts of the instruction word as well as the use of an algebraic assembly language rather than a mnemonic assembly language. This thesis describes the design and implementation of IPPALE, an Interactive PP Assembly Language Editor. IPPALE is a software tool that analyzes and displays real-time partial state information about assembly language instructions as they are being written. IPPALE provides information about syntax errors, instruction word conflicts, pipeline warnings, hardware resource usage, and also which resources are still available for use in the current instruction. This information allows the programmer to create very efficient code in a much shorter period of time.

Thesis Supervisor: Srinivas Devadas
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank the following people, whose assistance and support in creating this document were invaluable:

To Tomas Olson and the Image Understanding Group at Texas Instruments for their support, guidance, and assistance throughout this project.

To the DSP Group at Texas Instruments for the source code to *ppasm*, the PP assembler.

To Srinivas Devadas for his help in preparing the thesis document itself.

To Stephanie DeWeese, James Pearson, Tom Matusaitis, and James Brooks for helping me get the most out of MIT.

And finally, to my family for their continued support.

Contents

1 Introduction **9**

2 Architecture of TMS320C8x **12**

2.1 Overview of the TMS320C8x 12

2.2 Architecture of the Parallel Processor 15

2.2.1 Data Unit 15

2.2.2 Local Addressing Unit 15

2.2.3 Global Addressing Unit 17

2.2.4 Program Flow Control Unit 17

2.3 Programming the Parallel Processor 18

2.4 Difficulties Programming the PP 18

3 Design of IPPALE **21**

3.1 Goals for IPPALE 21

3.2 Overall Design 23

3.3 Editor Module 24

3.4 Engine Module 25

3.5 Graphic Display Module 27

3.6 Communication 28

4 Using IPPALE **30**

4.1 Starting IPPALE 30

4.2 Analyzing an Instruction 30

4.3 Interpreting the Results 32
4.3.1 Color 32
4.3.2 Assembly Instruction Block 37
4.3.3 Resource Usage Block 37
4.3.4 Instruction Class Block 38
4.3.5 Instruction Word Block 38
4.3.6 Message Block 39

5 Discussion **40**
5.1 Pros 40
5.1.1 Learning Tool 40
5.1.2 Many Different Representations 40
5.1.3 Little Required Effort 41
5.1.4 Useful Reference 41
5.1.5 *ppasm* Bug 41
5.2 Cons 42
5.2.1 Reliability 42
5.2.2 Speed 42
5.2.3 Adjustable Display Interface 43
5.2.4 Improved Editor Interface 43
5.2.5 Message Library 43
5.2.6 Integration 43

6 Conclusions **45**
6.1 IPPALE as an Educational Tool 45
6.2 IPPALE for Other Processors 45
6.3 IPPALE for Other Programming Languages 46

A Description of PP Instructions **47**
A.1 Instruction Classes 47
A.2 Instruction Subclasses 48

B Description of PP Instruction Word **50**

References **52**

List of Figures

2-1 Overview of the Architecture of the TMS320C80. 13
2-2 Block Diagram of the Parallel Processor. 16
2-3 Block Diagram of the Data Unit. 17

3-1 Overall Block Diagram of the IPPALE system. 23

4-1 Blank IPPALE Window. 31
4-2 Example IPPALE Display 33
4-3 Example IPPALE Display 34
4-4 Example IPPALE Display 35
4-5 Example IPPALE Display 36

List of Tables

2.1 Example Assembly Language Instructions for the Parallel Processor. . 18

2.2 Example Invalid Assembly Language Instructions. 19

A.1 Examples of Different Classes of PP Operations. 47

B.1 Instruction Bit Usage for Different Classes of Instructions. 51

Chapter 1

Introduction

Currently there are many differences between microprocessors and digital signal processors (DSPs). These differences are due to the fact that DSPs are generally used in applications which are much more computationally intensive than standard desktop computer applications. Some examples of these applications include video compression and decompression, object targeting and tracking, audio processing, and communications signal processing. DSPs are also often used in applications which much be accomplished in real-time [6].

Because of the differences in use between DSPs and microprocessors, DSPs have developed much different architectures from microprocessors. Two examples of this are substantial support for parallel processing and more complex control structures [6]. An example of the support for parallel processing is the TMS320C40, which offers six communication ports on each processor. After using two of these ports for instruction and data memories, we are still able to easily implement a two-dimensional array of interconnected processors [6]. Similarly, zero-overhead looping structures and extensive on-chip parallelism are some of the advanced control structures available on many DSPs.

These differences in architecture and use have limited the ways in which DSPs are programmed. Because speed is often extremely important, interpreted languages are almost never used. Because of the architectural complexity of the DSP, lower level languages, such as assembly language or C, are often the programming languages

of choice. Recently some DSP specific programming languages have started emerging, (DSP/C and Silage [6]) however these languages are still lower level compiled languages and similar in semantics and structure to C.

Because the complexity of DSPs is growing so rapidly, it is difficult for compiler technology to keep up. Therefore, compilers are becoming less efficient at reducing high-level code to efficient assembly language code. This is forcing many DSP programmers to resort to assembly language programming. This can cause much longer learning curves for each new DSP, as well as greatly increased product development times.

The lack of choices of programming languages, or programming "environments," is partly due to the history of programming languages. Traditionally, there have been two distinct types of programming languages: compiled or assembled languages and interpreted languages [9]. Compiled or assembled programming languages were similar in that the programmer entered all of the desired code into an editor. The code was then passed through a compiler or assembler which produced executable code. This is typically referred to as batch-style programming. Interpreted languages, however, allowed the programmer to enter code one line at a time. As the line is entered, it is evaluated and the state of the system is modified. Although interpreted languages offered much more user feedback, they usually created less efficient code.

Eventually programming environments like *Turbo Pascal* TM, *Turbo C* TM, and *ASMEdit* TM [4] incorporated the editor and compiler or assembler into a single program. Compiling code then required only entering a command key sequence. This improved software development times, however, it still completely separates the process of compilation from the process of entering code. It is the combination of these two processes that led to many of the advantages associated with interpreted programming languages.

There were also some attempts to change the representation of programs. Graphical programming environments allowed the user to use a mouse to select items, such as data paths. By selecting enough paths an instruction was created. Another programming environment allowed users to create graphical representations of assembly

language structures. These structures could then be used in programs and were later replaced by appropriate code [13]. Although these programming environments represented programs differently from the text-based assembly language, they still each only had one representation for programs. These environments also suffered from the fact that code often took longer to create than with text-based representations for very experienced programmers. The learning curve was much lower, however.

This thesis discusses a project which attempts to combine some of the advantages of interpreted programming languages with the requirements for DSP programming. The result is a programming environment which cannot be called a purely batch-style programming environment or a purely interpreted programming environment. This programming environment is called IPPALE, Interactive Assembly Language Editor for the Parallel Processor (PP) of the TMS320C8x. This thesis begins by first describing the DSP for which this programming environment is designed, the PP of the TMS320C8x (Chapter 2). Next, it describes the design of the IPPALE system (Chapter 3). Then it describes the resulting programming environment and what it is like to use it (Chapter 4). Finally, it discusses the benefits and problems associated with the final system and the implications for other processors as well as other programming languages (Chapters 5 and 6).

Chapter 2

Architecture of TMS320C8x

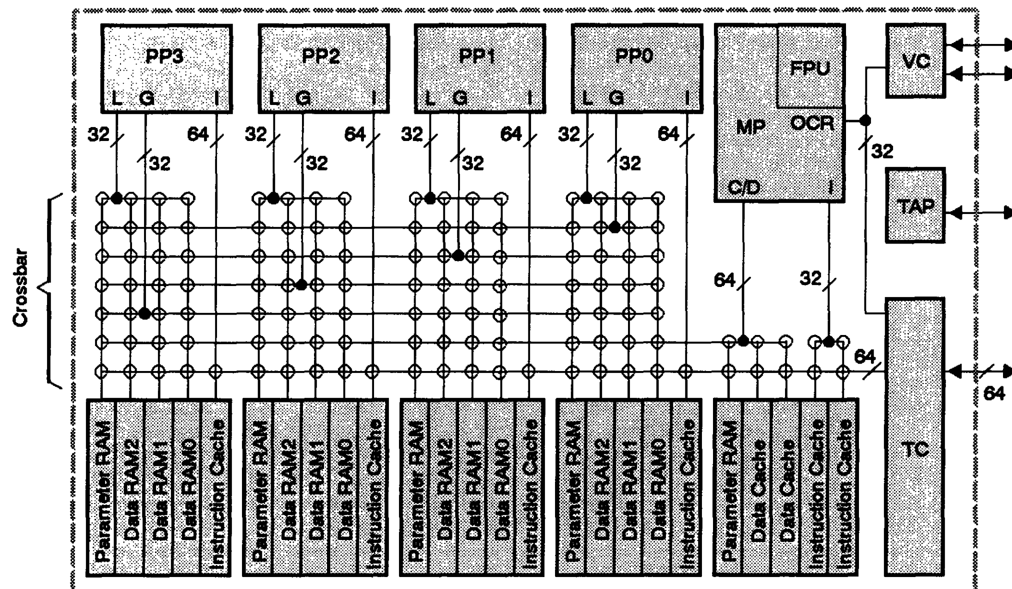
This chapter describes the architecture of the TMS320C8x family of DSPs from Texas Instruments, focusing on the architecture of the TMS320C80 and the architecture of the Parallel Processor. It also discusses the tasks involved in programming the TMS320C8x family of DSPs, including the tasks required to program the PP. More detailed information about the architecture and programming of the TMS320C8x can be found in the *Texas Instruments TMS320C80 Technical Brief* [16].

2.1 Overview of the TMS320C8x

The TMS320C8x is a new line of DSPs offered by Texas Instruments. This family of processors currently includes the TMS320C80 and TMS320C82. Both of these processors are single-chip multiprocessor DSP "solutions" or systems. They both contain one 64-bit RISC Master Processor (MP). The C80 contains four 32-bit Parallel Processors(PPs) and 50 kilobytes of on-chip RAM, while the C82 contains only two PPs and 44 kilobytes of on-chip RAM. Both processors also contain a very fast crossbar which potentially allows all of the processors to access data simultaneously and a Transfer Controller(TC) that takes care of all off-chip memory accesses. A block diagram of the C80 can be seen in Figure 2-1.

Programming the C8x involves basically four tasks. These tasks are:

- Initiate Data Transfers with the Transfer Controller



Note:

L	=	Local port	G	=	Global port
I	=	Instruction port	FPU	=	Floating-point unit
OCR	=	On-chip register port	C/D	=	Cache/data port
TAP	=	Test access port	TC	=	Transfer controller
VC	=	Video controller	MP	=	Master processor
PP0–3	=	Parallel processors 0–3			

Figure 2-1: Overview of the Architecture of the TMS320C80.

- Setup and Initiate the Parallel Processors
- Manage the Parallel Processors
- Program the Parallel Processors

The first three of these tasks are usually executed by the Master Processor. The MP is generally programmed in C because very efficient compilers are relatively easy to produce for RISC processors.

Setting up the PPs is a relatively simple process. It involves loading a starting address for the code to be executed into the correct register for each PP. Then the Master Processor can send a signal to any or all of the PPs to force them to start executing code. The PPs will then continue to execute code until they receive a halt command. Either the MP or the PP itself can issue a halt command.

The TC can either be setup by one of the PPs or by the MP. The data memory of the PPs must be explicitly loaded from external memory by the TC. Often the MP executes the code necessary to set up the TC to pre-load the data memory of the PPs. The PPs may execute code which causes the TC to load or store additional memory once they have begun executing code.

For multitasking systems, the task of managing the PPs during operation becomes much more difficult than for systems which always have the PPs running the same piece of code. This type of process management is often performed by pre-existing multitasking software. An example of this software is Texas Instruments' *Multitasking Executive* [16].

The last of the four tasks involved with programming the C8x is programming the PPs. Of the four tasks, this is usually the most difficult and time-consuming. This is partially due to the fact that each C8x series chip has multiple Parallel Processors which can be executing different code simultaneously and also due to the fact that each PP is very complex. Programming the PP can be done in C, however, it is almost always done in assembly language for efficiency. It is only this task of programming a single PP that IPPALE is designed to make easier.

2.2 Architecture of the Parallel Processor

A block diagram of the PP can be seen in Figure 2-2. Each PP of the C80 or C82 consists of four major functional units:

- Data Unit (DU)
- Local Addressing Unit (LAU)
- Global Addressing Unit (GAU)
- Program Flow Control Unit (PFCU)

2.2.1 Data Unit

The Data Unit(DU) performs most of the computational work within the Parallel Processor. It contains a 16x16 bit multiplier (which is splittable into two 8x8 bit multipliers), scaling and rounding logic for the multiplier, a barrel rotator, bit detection logic, a mask generator, an expander, and a three-input ALU. A block diagram of the DU can be seen in Figure 2-3.

Although this diagram of the DU shows many different datapaths which are available, any arbitrary combination of datapaths is not permitted. This is because the instruction word is not long enough to fully specify an arbitrary combination of datapaths within the DU in addition to addressing and control information.

2.2.2 Local Addressing Unit

The Local Addressing Unit(LAU) performs addressing operations using the local data port(the L port in Figure 2-1). The LAU contains an ALU, a scaling unit for address calculation, and hardware for field extraction. Figure 2-1 shows that the L port can only access one major block of memory, the block associated with the PP. If an address in calculated at run-time to be outside of this block, the processor is stalled and the G port is used during the stall cycle.

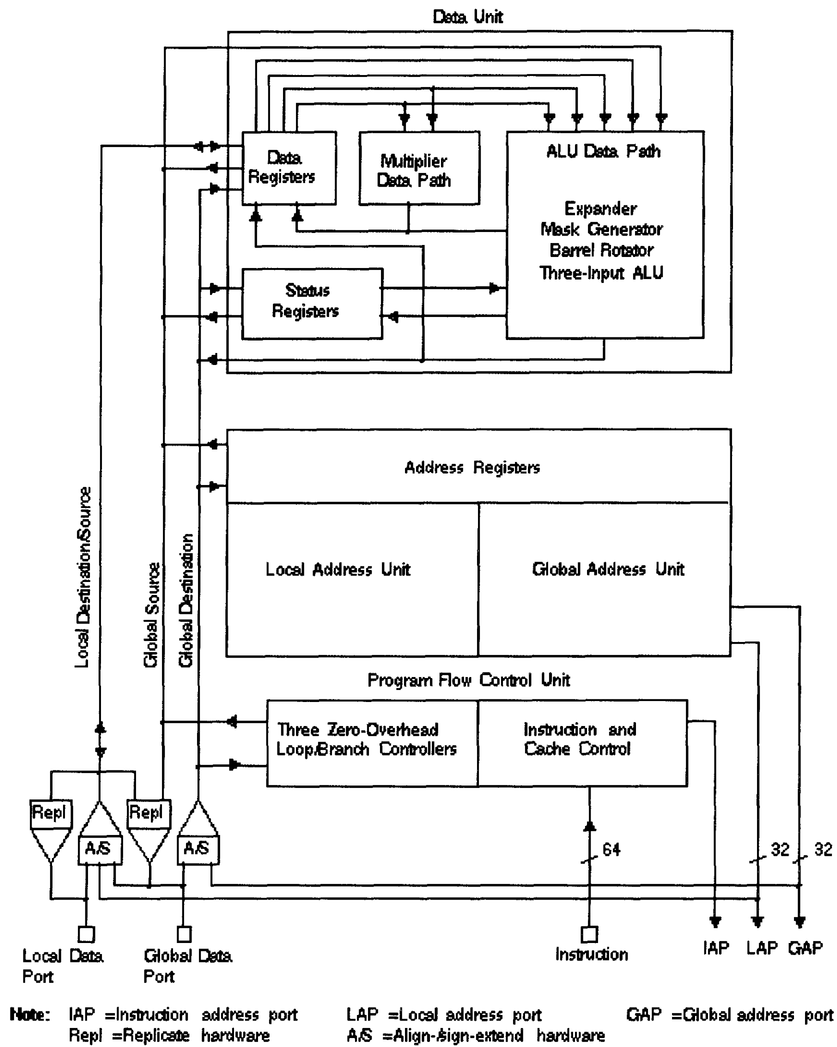


Figure 2-2: Block Diagram of the Parallel Processor.

Figure 2-3: Block Diagram of the Data Unit.

2.2.3 Global Addressing Unit

The Global Addressing Unit(GAU) performs addressing operations through the global data port(the G port in Figure 2-1). The GAU contains all of the hardware that the LAU has. The G port, however, can access any of the four major memory blocks associated with the four PPs on the chip. This is assuming that no two PPs are trying to read the same slice of memory. Memory address conflicts of this type, like the improper addresses in the LAU, are dealt with at run-time and involve stalling a processor.

2.2.4 Program Flow Control Unit

The Program Flow Control Unit(PFCU) is the last major functional unit within the Parallel Processor. The PFCU controls the execution of instructions within the PP. It controls up to three levels of zero-overhead looping, the instruction cache controller, and updates the registers that contain the instruction word for each stage of the

```
d1 = (d1 & d3) + (0x32 & d3)     ; ALU Operation
*(a1 + x2) = d4                  ; Store Operation
call =[nz] mult                  ; Conditional Call
d5 =[z] d4 * d3 || d3 =[z] d2 + d1>>-d0 ||
        d2 = *a15                ; Parallel Operation
```

Table 2.1: Example Assembly Language Instructions for the Parallel Processor.

pipeline.

2.3 Programming the Parallel Processor

As described earlier, the PP is almost always programmed in assembly language. The assembly language used, however, is not a standard mnemonic based assembly language. Instead, the PP uses an algebraic assembly language. Because of the large set of possible instructions, mostly in the DU, a mnemonic assembly language would have needed literally thousands of distinct operation symbols. This would have led to extremely long operation symbols, unreadable code, and a very long learning period for beginning programmers. To avoid this, TI decided to use an algebraic assembler. Some examples of legal commands are shown in Table 2.1.

This algebraic assembly language offers several advantages over more traditional mnemonic assembly languages, especially for this particular application. The most important advantage is the improved readability of code. With a very short introduction to this assembly language, many people can scan through sections of code and understand, at least at the instruction level, what is happening.

2.4 Difficulties Programming the PP

Although a comparison between legal mnemonic assembly language code and legal algebraic assembly language code shows that the algebraic code is more readable, there are disadvantages to algebraic languages. The main disadvantage comes from the reduced amount of structure built into the language. It is easy for a programmer

```
d1 = d1 + d2 + d3
d1 = d2 / d4
d1 = d2 * d3 || d5 = d7 & 0x3
d1 = (d4 % d5)//d2
```

Table 2.2: Example Algebraic Expressions which are Not Valid PP Assembly Instructions.

to determine legal operations in a mnemonic assembly language. If a symbol for the operation type exists, then it is a legal operation. The user doesn't need to understand the intricacies of the hardware so long as they accept the fact that the list of instruction mnemonics represents the only legal instructions that can be executed. Understanding the possibilities and limitations of the processor consists solely of memorizing all of the legal operation symbols.

In an algebraic assembly language, however, there are an infinite number of commands that conform to the lexical and syntactic rules of algebraic expression. For various reasons, however, not all of these instructions are legal as PP instructions. Table 2.2, for example shows several different illegal PP instructions which conform to the rules for algebraic languages.

Each of the instructions in Table 2.2 is illegally for a slightly different reason. The first instruction tries to use an ALU operation which the ALU does not support. Although the PP does contain a three input ALU, only operations of the following form are valid instructions:

```
dst = A & f1(B,C) + f2(B,C) [+1 | +cin] ; A, B, \& C are inputs to the
                                         ;   three ports of the ALU
                                         ; f1 \& f2 are boolean combinations
                                         ;   of B and C.
```

The second instruction in Table 2.2 cannot be computed because the PP does not contain hardware for a single cycle divide. In fact, there is a PP command which can be used to assist in computing division, however, this command must be used within a loop. Essentially, this command uses the multiplier and the ALU (for subtraction). This command will not, in general however, produce an answer within a single cycle [18].

The third instruction in Table 2.2 cannot be computed because the 64 (or even 96) bits in the instruction word are not enough to specify this many operands or operations. In fact, when a multiply operation is being performed, no immediate value can be used in either the multiply or ALU portion of the instruction because of the lack of instruction word bits.

The fourth example in Table 2.2 cannot be computed because the PP does not have datapaths which would allow this. Figure 2-3 shows that the output of the Mask Generator cannot be fed back into the input of the Barrel Rotator during the same cycle. Therefore this instruction cannot be computed in a single cycle.

There is nothing within the structure of an algebraic language that makes any of these restrictions true. Only the hardware of the PP creates these constraints. Constraints like these appear for many reasons and in many ways in the PP of the C8x series of DSPs.

These differences between what seems legal and what is legal come from using a previously existing virtually unlimited language to describe a limited set of possible instructions. In order for the programmer to use this language, he or she must memorize a list of rules concerning what types of operations can be done, what combinations of these operations are allowed, and how they are represented in the language. Conventional assembly languages generally use a limited number of mnemonic symbols to encompass the above rules. Representation, or the list of symbols, then becomes the only rule the programmer needs to learn.

A brief description of some of the rules necessary for programming the PP are given in Appendix A. Appendix B describes the usage of the bits in the instruction word. Appendix C includes a partial list of tables which would be useful to have available while programming the PP. Even advanced PP programmers need many of these tables while programming the PP. From these tables it becomes clear that a programmer must view each instruction in many different ways in order to ensure that it is legal. IPPALE attempts to assist the programmer by representing the instruction in ways which clarify the different types of instruction word interactions described above.

Chapter 3

Design of IPPALE

3.1 Goals for IPPALE

The initial motivating factor for the IPPALE project came from the fact that the batch-style method of programming, along with the constant referencing of tables greatly reduced the productivity of programmers. In order to verify instruction validity, the programmer would have to enter the instruction using an editor, save the file, run the assembler, *ppasm*, on the file, and then determine from the output of *ppasm* whether or not the instruction was legal. If the programmer then wanted to change the instruction to pack more computation into a single instruction, the entire process would have to be repeated.

The overall goals of the IPPALE project were to:

- Facilitate learning to program the PP
- and provide additional real-time information which would increase the productivity of experienced programmers.

If successful, this would greatly reduce the number of tables from Appendix C that typical programmers need to reference regularly while programming the PP. These two goals were broken down into five more specific functions that IPPALE should perform[1].

First, IPPALE should display, in an quickly understandable manner, exactly what the current instruction is doing and what resources it is using. It should do this quickly because

[1]These five functions were chosen based on [2], which describes specific user engineering principles for designing interactive systems.

the programmer will be using this information while programming. In order for the programmer to remain efficient, the most relevant information should be available with a short glance. This function serves to teach the novice programmer which resources each type of instruction uses, as well as verify to the experienced programmer that the current instruction is valid.

Second, IPPALE should display what the current instruction could still do. This includes both additional operations that can be done in parallel, as well as additional features or functional blocks that can be used in combination with what is already being used. This helps teach novice programmers what combinations can and cannot be used together. It also means that experienced programmers don't have to spend as much time looking through reference tables and charts to try to find legal combinations for infrequently used, unfamiliar operations.

Another major function of IPPALE is to assist the programmer in determining why an illegal instruction is illegal and what could be done to make it legal. This would include dealing with both illegal instructions and illegal combinations of legal instructions. The extreme of this goal would be for IPPALE to produce a text explanation which tells programmers what is illegal about the current instruction and what can be done to fix it. One example of the extreme of this goal would be something like the programmer's assistant [10]. Instead, what IPPALE does is try to draw attention to possible reasons for conflict. It would then be up to the programmer to use these hints to determine exactly why the instruction is illegal.

All of the information which IPPALE provides so far is relevant only to the current instruction. Although the focus of this tool is on representing the effects of a single instruction, there are several inter-instruction conflicts which are peculiar to the PP. It would be nice if IPPALE could give some warnings about when these inter-instruction conflicts may arise. One example of this is the fact that a write to an "a" register is not available to the addressing unit during the next instruction. This is due to the pipelining of the PP. Other useful warnings could include information about branch delay slots, and the effect of performing a conditional load using an address register which is modified [18]. [2]

The last specific goal that IPPALE must satisfy is that it should not interfere with or inhibit the programmer in any way. Using this tool should not require any more effort or

[2]It turns out that the address register is *unconditionally* modified even though the load is conditional.

attention than using a simple editor. There are several implications to this. If IPPALE were to crash, the programmer should not lose the code that had been written. Also, the programmer should never have to wait for IPPALE while writing code. IPPALE should not be distracting to the programmer, but only an aid that is available when desired. Although the novice programmer may accept some distractions in order to become more familiar with the PP, the more experienced programmer would cease to use a tool that interferes with his/her productivity. If this tool is to be used by experienced programmers, it cannot be a nuisance in any way.

3.2 Overall Design

The overall system design of IPPALE was largely influenced by the fact that IPPALE should not interfere with the programmer. In order to accomplish this, it was decided that the editor and the rest of IPPALE would be two separate processes with minimal communication. This allows the user to use the editor as fast as possible without worrying about the rest of IPPALE. Since they are separate processes, the rest of IPPALE is executed in parallel with the editor and does not necessarily have to keep up with it.

In order to make IPPALE easy to maintain and upgrade, it was further broken down into the three main modules shown in Figure 3-1. These modules make it easier to upgrade one element of the system without upgrading the entire system. Breaking IPPALE into three distinct components also allows us to take advantage of the fact that each of the three main functions of IPPALE was best suited to a different programming language.

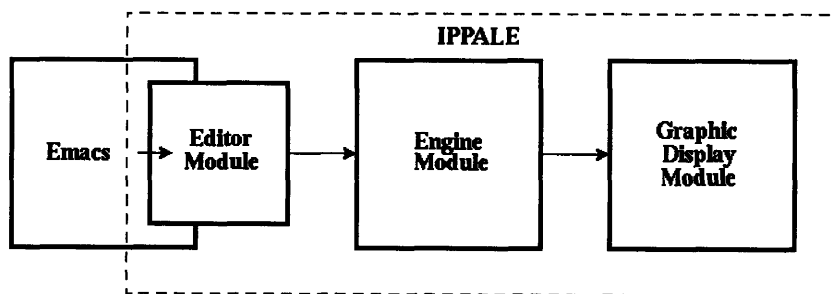


Figure 3-1: Overall Block Diagram of the IPPALE system.

The Editor Module is actually embedded within a standard text editor and determines

what instruction will be analyzed and displayed by IPPALE. The Engine Module decomposes the algebraic assembly language text and determines, to the best of its ability, exactly what is happening within the PP during that instruction and some of the possible consequences for preceding instructions. This information is then used to determine exactly what should be displayed in order to represent what is happening. The Graphic Display Module displays the information in a way that draws attention to the most important information and de-emphasizes the less important, but still useful, information.

It is important to note that information only flows in one direction in this system. Because of this, any crash or bug in IPPALE can never affect the code that the programmer has typed into the editor. This is because there is no way for information to flow backwards and influence the editor.

3.3 Editor Module

The Editor Module serves two purposes within IPPALE. First, it provides a link between the text editor used to enter assembly language code and the rest of IPPALE. Second, it must be able to determine and send relevant information at the appropriate time. Because it was not clear exactly what information would be relevant and even less clear when the appropriate times to send information were, the editor that we chose needed to be easily modifiable. Writing a new editor from scratch would require a significant amount of time and resources which would be better used on the rest of IPPALE. Because of these two requirements, using EMACS as the base of the editor seemed a natural choice. The EMACS editor is actually implemented within a programming language(EMACS-LISP) [15]. Because EMACS provides an interface with the code which controls the editor, making changes to the EMACS editor is relatively simple.

The E-LISP code written to modify EMACS consists of two major parts. The first group of functions make it possible to create, terminate, and send information to, a separate process which will be the rest of IPPALE. This is accomplished using the library of E-LISP functions for asynchronous processes [5]. These are processes which are started by EMACS itself, but run independently from, and in parallel with, EMACS.

The second group of functions tries to fulfill the goal of sending relevant information. Since a parallel instruction (or even a single instruction) can be spread over several lines,

this code must determine where an instruction begins and where it ends. Also, due to the rules for communication between the Editor Module and the Engine Module (see Section 3.6), a line of assembly language code that is sent to the Engine Module cannot have comments anywhere within it.[3] Finally, while checking for and removing comments, the Editor Module also compresses whitespace in order to reduce the amount of unnecessary communication between modules.

Currently the Editor Module does not attempt to determine when it is appropriate to send information. Instead, the code that extracts and sends a legal instruction is bound to a key sequence. It is the the user's responsibility to determine when information should be sent.

3.4 Engine Module

The Engine Module serves as the heart of IPPALE and includes more than 75 percent of the code. There were two basic approaches that could be taken to create this assembly language "interpreter." The first approach was to create a parsing engine that could apply all of the rules governing the algebraic assembly language of the PP. This would be somewhat difficult because the "rules" governing valid PP instructions are based on hardware constraints rather than any type of language syntax. We would be trying to implement rules about specific hardware constraints as language rules. This does not seem to be the "natural" representation of the rules.

The second approach was to use assembler/interpreter techniques to transform the code into the actual instruction word bits and extract the relevant information from the instruction word. This information would be then displayed using several representations. The advantage to the second approach is that many of the conflicts in parallel programming of the PP come from not having enough bits in the instruction word. It is often the case that much more information can be easily obtained from the instruction word bits than the text assembly instruction.

In order to ensure compatibility between what IPPALE says is legal and what *ppasm* says

[3]This is because an instruction going into the Engine Module is sent on a single line. Since the terminating symbol for comments in the PP assembly language is a newline character, a comment in the middle of an instruction would cause the Engine to ignore the rest of the line.

is legal, we decided to use the source code for *ppasm* as part of the Engine Module. Several modifications were needed in order to make this assembler work more like an interpreter. Some of these modifications include:

- Make *ppasm* use standard input rather than read a file (which means it can only look at any particular data once)[4]
- Make *ppasm* determine all possible ways of encoding instructions rather than only finding the first legal one
- Make *ppasm* do as much of the encoding as it can on illegal instructions rather than quitting after the first error
- and make ppasm deal with undefined symbols/labels in an intelligent manner.

This modified assembler attempts to encode each parallel part of each instruction in every way possible. After each encoding, the encoded information is passed to a new group of functions built around the assembler. These functions determine and keep track of resource usage information and update the information with each new encoding possibility received.

In order to accomplish this, the Engine Module creates resource lists, which associate attributes with each resource. There are five different attributes in the IPPALE system:

- free
- used
- conflict
- blocked (or banned)
- multiple possibility (or possibly used)

If a resource is free, there is some way that this resource can be used in combination with *every* resource that is currently being used. If a resource is used, then there is one, and only one, part of an instruction which requires this resource in order to execute. A conflict in a resource occurs when one part of an instruction needs to use a resource, but another part of

[4]This is more involved that it first appears. Most assemblers work by making several passes through the entire file before any output is created. This is not possible with our slightly interpretive version of the assembler. All analysis has to be performed in a single pass.

an instruction needs to use it also or block it. A blocked resource is a resource which cannot be used with all of the current resources which are used. This is often because not enough bits exist in the instruction word to use the resource. A resource is a multiple possibility if there are several ways to encode an instruction, and at least one, but not all, encodings use the resource.[5] Examples of these attributes are given in Chapter 4 with the description of the IPPALE programming environment.

The Engine Module actually keeps two resource attribution lists. One list is used to represent the attributes of resources for all of the previous parallel instruction parts. The other list represents the attributes based only on the current parallel part of an instruction. For each possible encoding of the current parallel part of an instruction, the resource list for this part is updated using a finite state automaton(FSA). The FSA uses the attributes as states. For example, if the current "state" of a resource is used, an another possible encoding of the instruction says that a resource will be free, the resulting state is multiple possibility. This is because in one encoding the resource is used, however, in another encoding, it is not needed.

The FSA for each of the two lists is also different. This is because one list updates the resources based on several possible, mutually exclusive, encodings. The other list, on the other hand, combines several necessary combinations of resource lists. In other words, this list represents, in some sense, the union of the lists for each parallel part of an instruction. Because of this difference, the rules for changing states are different for the two different lists.

When all of the parallel parts of an instruction have been encoded, the Engine Module sends the attribute lists, as well as some other information, to the Graphic Display Module. Then, all of the lists are cleared and prepared for the next instruction.

3.5 Graphic Display Module

At a high level, the Graphic Display Module's job is very easy. In essence, the Engine Module tells the graphic display, "Print the assembly instruction 'd1 = d2 + d3,' Color the Multiplier Resource Yellow, Color the ALU Resource Green, ..." In order to reduce the amount of communication, however, this long string is compressed. The resource being

[5]Note that if all possible encodings of an instruction use a resource, it is considered used. Similarly, if all possible encodings block a resource, it is considered blocked.

described is determined by the location within the string being read. The character at that location represents one of the five attributes or states described above.

The job of the Graphic Display Module is to read in a list of attributes, apply them to the resources they correspond to, and update the display. The Graphic Display Module uses one important lesson learned from EMACS, also. The reading of information has a higher priority than updating the display [15]. This means that a large burst of information sent to the Graphic Display will only display the final set of information.

This means that a programmer can type incredibly fast, sending instructions to the Graphic Display faster than it can keep up. When the programmer stops typing, (because he wants to see how an instruction is encoded, for example) the Graphic Display will not cycle through each instruction that had previously been typed, but only display the most recent information. This prevents the programmer from waiting for the Graphic Display to "catch up."

All of the display routines are implemented using the tkinter module in the Python programming language [19]. This module of Python provides a very clean, object-oriented interface to the Tk toolkit.

3.6 Communication

Although using a different programming language for each of the three modules eases the implementation of each module, one aspect that is made more difficult is the interface between the modules. For IPPALE, the UNIX pipe is the primary method of communication. In order to use this type of communication, we needed to establish some rules for communication. This section describes the rules for communication between the three different Modules.

The communication between the Editor Module and the Engine Module is very simple. The Editor Module must be able to send the current assembly language instruction to the Engine Module. Only one communication rule was needed between these two modules. Each complete instruction, including all of the parallel parts, are to be sent on a single line. In other words, a newline character is the termination symbol for the current instruction.

The rules for the communication between the Engine Module and the Graphic Display Module were briefly described above. In fact the first character in a string represents the Block (see Section 4.3) for which the information is relevant. Following this is a list of

characters. The characters used are 'u' (used), 'f' (free), 'c' (conflict), 'b' (blocked), and 'm' (multiple possibility). The order of the characters determines which resource gets which attribute.

The Engine Module can also send messages to the Graphic Display. These are sent as strings with an introductory character that designates a message.

Chapter 4

Using IPPALE

This chapter describes the IPPALE programming environment. It discusses what is required of the programmer in order to use it, as well as what the information displayed represents.

4.1 Starting IPPALE

In order to use IPPALE the programmer must first start an EMACS session. Next, the E-LISP code for the Editor Module should be loaded into EMACS. This can either be done automatically by adding a line to the programmer's ".emacs" file, or it can be loaded manually. Once loaded, the programmer enters a command-key sequence to start IPPALE. Loading and initializing IPPALE can be time consuming. It can take as long as 10 seconds on a fast machine with few other processes or as long as 30-45 seconds on a slower machine with many processes already in progress. The initialization opens a new window which has several different representations of an assembly instruction. Figure 4-1 shows the IPPALE window immediately after it has appeared.

4.2 Analyzing an Instruction

Once started, the programmer can, at any time, send the instruction that includes the cursor to IPPALE. Currently, sending the instruction is done by entering a command-key sequence to EMACS. After entering the sequence, the IPPALE display is updated. The display is usually updated within 1-2 seconds on a relatively fast machine. Some example display

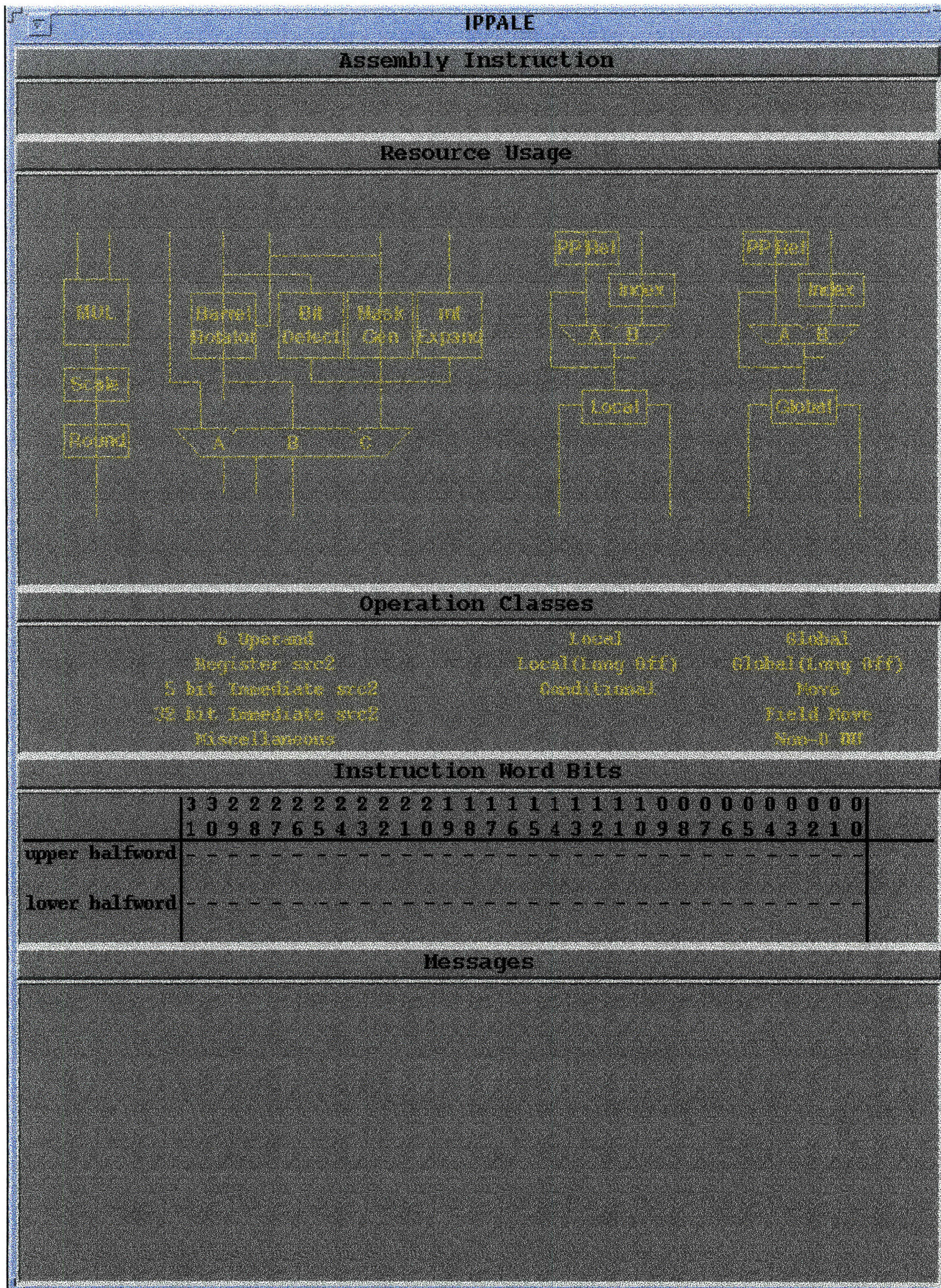


Figure 4-1: Blank IPPALE Window.

displays are shown in Figures 4-2, 4-3, 4-4, and 4-5.

4.3 Interpreting the Results

The IPPALE window has 5 different display "blocks". Each of these blocks presents a different representation of the current assembly language instruction. IPPALE uses colors and symbols within each block to present the hardware 'meaning' of the instruction in several ways.

4.3.1 Color

Although it cannot be seen in the previous figures, color is used extensively in IPPALE. Different colors are used to quickly draw the attention of the programmer to certain aspects which may be important. There are five main colors used in IPPALE: red, yellow, gray, black, and green.

Red is used to quickly draw the programmer's attention to potential problems. Red represents resources which are assigned the attribute of "conflict" described earlier. When the attribute conflict is assigned to a resource, this means that IPPALE, and therefore the *ppasm*, have no way of legally encoding the current instruction. Coloring the resource red alerts the programmer to the mistake immediately and draws the programmer's attention to likely areas of explanation.

Yellow is used to represent "free" resources. Resources which are not yet used, but which could still be used in the current instruction are 'highlighted' in yellow.

Gray is used to represent "blocked," or unavailable, resources. On a dark gray background, the use of gray does a good job of 'hiding' the resources which are unavailable. Although they are still slightly visible, the programmer's attention is strongly drawn away from these resources.

Black is used to represent resources which are currently being used. Unlike each of the previous colors, black is very dark. This helps to strongly distinguish black, or "used," resources from other resources. This is especially useful for novice programmers who are learning which resources different instructions require. Since black is not bold, however, it becomes very easy to ignore as the programmer becomes more interested in other types of resource attributes.

Figure 4-2: IPPALE Display Window Showing a Legal PP Assembly Language Instruction.

IPPALE

Assembly Instruction

```
d1 = d2 + d3
|| d4 = *a10
```

Resource Usage

Operation Classes

6 Operand — Local — Global

Register src2 — Local(Long Off) — Global(Long Off)

5 bit Immediate src2 — Conditional — Move

32 bit immediate src2 — Field Move

Miscellaneous — Non-D DU

Instruction Word Bits

```
               |3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0|
               |1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
upper halfword |1|class|A| ALU Operation | dst |src1 |1 0|-|src2 |- - - - - - -
               |1 1 1 1 1 0 1 0 1 0 1 0 0 0 0 1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0|
lower halfword |- - - - - - -|Gim/X|-|Gbank|L| Gmode | reg |e|siz|s| Ga |- - -
               |0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 0 1 1 0 1 0 1 0 0 0 0|
```

Messages

```
The following are possible encodings for each part of the instruction:
fa81710000000000
8a21518000000000
fa81518000000000
8a21710000000000
0000000000135350
```

Figure 4-3: IPPALE Display Window Showing a Legal Combination of PP Assembly Language Instructions.

IPPALE

Assembly Instruction

```
d4 = *a1 || d1 = d2 + d3
 || d2 = d4 || d3 = *a10
```

Resource Usage

Operation Classes

Local

Global

Register src2

5 bit Immediate src2

Move

Instruction Word Bits

```
               |3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0|
               |1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
upper halfword |1|class|A| ALU Operation | dst |src1 |****m*src2 | Lmode |  d  |
               |1 1 1 1 1 0 1 1 1 0 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 0 1 0 1 0 0|
lower halfword |e|siz|s| La   |**m/X|***bank|L|*******| *** |*******| *** |Lim/X|
               |1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0|
```

Messages

```
fa81518000000000
8a21518000000000
fa81710000000000
0000000000101110
8b22800000000000
0000000000134f50
```

Figure 4-4: IPPALE Display Window Showing an Illegal Combination of Legal Assembly Language Instructions.

IPPALE

Assembly Instruction

d2 = d4

Resource Usage

MUL
Scale
Round
Barrel Rotator
Bit Detect
Mask Gen
mf Expand
A B C
DP Rel
Index
A B
Local
DP Rel
Index
A B
Global

Operation Classes

6 Operand
Register src2
5 bit Immediate src2
32 bit Immediate src2
Miscellaneous
Local
Local(Long Off)
Conditional
Global
Global(Long Off)
Move
Field Move
Non-D DO

Instruction Word Bits

	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
upper halfword	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
lower halfword	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Messages

Figure 4-5: IPPALE display window showing an assembly language instruction which can be encoded in multiple ways.

Green is used for "multiple possibility" resources. Green was chosen for several reasons. One reason is because it is dark. Because "multiple possibility" resources are similar to used resources, a darker color seemed useful. Since "multiple possibility" instructions provide many options, as well as causes for concern in programs, drawing the programmer's attention to these instructions seemed important. This is also why green is an excellent choice for this type of resources.

4.3.2 Assembly Instruction Block

The first block in the IPPALE display is the Assembly Language Instruction Block. This block displays the instruction that was last sent to the Engine Module of IPPALE. This window can be used to verify that the instruction received is the intended instruction. For example, this block would help a programmer determine that the parallel symbol, "||," was accidentally left out of an instruction.

Since this block does not present extremely useful information, it is small compared to the other blocks. It is also placed at the top edge of the window, leaving the center of the window for the more important information. This block also does not use color. This keeps attention from being drawn to it unnecessarily.

4.3.3 Resource Usage Block

The second block is one of the most important. It is the Resource Usage Block. It displays all of the data paths and functional units within the PP in the form of a block diagram. Each of these resources is given a color to designate how it is being used in the current instruction.

This block represents the current instruction at the hardware level. This block is very useful for introducing beginning programmers to the hardware and functionality of the PP, and it is also a useful way to view the subclass information for ALU instructions.[1] Since the subclasses of ALU instructions are based on the different possible input paths to the ALU ports, this block most clearly displays the current usage and additional possible usage.

[1]See Appendix A for more information about PP instruction classes and subclasses.

4.3.4 Instruction Class Block

The third block of the Graphic Display Module, the Class Block, displays the instruction information in a different form. There are 13 different instruction classes within the PP.[1] The Instruction Class Block shows all of the 13 instruction classes. In addition, each column within the Class Block represents a set of mutually exclusive classes. That is, at most, one instruction class can be used from each column.

Again, color is used to map attributes to instruction classes. This block is very useful for several reasons. Although many of the rules regarding combinations of classes are relatively simple, there are a few strange cases. This block can assist the programmer by showing the programmer which instruction classes are no longer available given a particular set of class choices.

This block also quickly draws attention to problems involving instruction class conflicts. In Figures 4-2, 4-3, and 4-4 you can see that each column has one or fewer classes with used attributes. Conversely, in Figure 4-5 you can see that one of the columns has two instruction classes with the attribute of "conflict." The reason this instruction is not legal is because the two parallel instructions are trying to map into two mutually exclusive instruction classes. Although not explicitly stated in this form, the red instruction classes draw the programmer's attention to this fact.

4.3.5 Instruction Word Block

The fourth block, the Instruction Word Block, is not very informative to the novice PP programmer. It is, however, informative to the more advanced programmer. This block allows the programmer to see all of the fields within the 64 bit instruction word. Sometimes this may reveal additional functionality that the programmer was otherwise not aware of. For example, the programmer can quickly refresh his/her memory on the maximum allowed length of immediate values. Another example is in Figure 4.a. The "siz" block can remind the programmer that loads and stores can also be done using bytes or halfwords in addition to using full 32 bit words.

In addition to providing insight into additional functionality, this block is the only way to understand why a few combinations of instructions are not allowed. According to the *TMS320C80 Parallel Processor User's Guide*, a PP-relative Global Addressing Unit Opera-

tion is in the same class as any other normal Global Addressing Unit Operation. Most Global Addressing Unit Operations can be done in parallel with a Local Addressing Unit Operation, however, a PP-relative Global Addressing Operation cannot be executed in parallel with a Local Addressing Unit Operation. This is not because data paths do not exist for this combination of instructions. It also does not violate the rules for the instruction classes. The only conflict is in the instruction bits. The bits in conflict have red stars in their spots. The other bits, which are not in conflict, are labeled with the appropriate fields.

In fact, if the programmer were to enter each of the above two operations separately, it would be clear why those two operations cannot be executed in parallel. There are extra instruction word bits needed for Global PP-relative addressing operations which are incompatible with LAU operations. Similarly, the bits used to specify Local PP-relative addressing operations are incompatible with GAU operations. This, along with several other limitations, are only clear when looking at the actual instruction word bits being used.

4.3.6 Message Block

The final block, the Message Block, is where all text messages from IPPALE are displayed. Currently, only warnings and error messages from the *ppasm* are displayed in this block. The library of messages displayed can easily be enlarged, however, to include warnings about some inter-instruction interactions, as well as more information about illegal instructions.

Chapter 5

Discussion

At the time of writing this thesis, the first version of IPPALE had just been completed, therefore there is very little feedback. The observations in this chapter are based on the author's experience with IPPALE, in addition to the comments of several DSP programmers (though not all C8x programmers) after a brief presentation and demonstration of the tool.

5.1 Pros

5.1.1 Learning Tool

IPPALE seems to be a useful learning aid for the PP. IPPALE adds structure to the algebraic assembly language of the PP. This structure helps the novice programmer learn more quickly what the PP can and cannot do in a single instruction. In addition, the learning curve is accelerated because the feedback about instructions is instant. This real-time feedback, in contrast to the batch-style feedback of typical assemblers, allows the programmer to experiment with many more instructions in a given amount of time. It also allows the programmer to experiment for a longer period of time before losing interest.

5.1.2 Many Different Representations

One of the great advantages of IPPALE is that it displays the current instruction information in more than one representation [8]. Often one particular representation explains how parallel instructions interact, while other representations do not explain the interactions. Seeing all

of the different representations simultaneously allows the programmer to see how instructions interact in certain representations, but not in others.

This is also especially useful for novice programmers. One particular representation may be easier to understand. The novice can experiment with different instructions paying particular attention to the representation that makes sense. Gradually, the relationship to other representations will begin to be understood.

5.1.3 Little Required Effort

Another advantage of IPPALE is that it provides all of its information at a very small cost to the programmer. Currently, the programmer is only required to start IPPALE and enter a command whenever an instruction should be analyzed. There is no cost for instructions which do not need to be analyzed. Even if IPPALE crashes, the editor and the code written by the programmer remain intact. These factors make IPPALE a useful tool, even if it is only rarely used, as would be the case with experienced programmers.

5.1.4 Useful Reference

There are approximately 20 tables and figures in Appendix A and Appendix D of the *Parallel Processor User's Guide* [18] which would be used regularly, even by experienced programmers, when programming the PP. Of these 20 tables IPPALE provides the information contained in almost half of them. This means that the programmer will need far fewer tables available as code is being written. Since far fewer tables are used regularly, the amount of time before the information becomes memorized will be much less using IPPALE.

In addition, IPPALE provides only the information from these tables which is relevant to the current instruction. Being able to filter out a large part of the information allows the programmer to focus on the higher level, more complicated tasks of programming such as the general flow and organization of code rather all of the details of each instruction. The details are summarized and displayed by IPPALE.

5.1.5 *ppasm* Bug

Finally, IPPALE actually drew attention to a bug in the *ppasm*. During a test of IPPALE, the author noticed that there were resources which were marked available, but in fact, were not

supposed to be. Tracing the problem led to the Engine Module, the *ppasm* code. Although the bug was only caused by a relatively obscure combination of commands[1], this demonstrated the benefit of looking at instructions in many different ways. Many illegal instructions become obvious when displayed by IPPALE, but would never be obvious when looking only at the algebraic representation.

5.2 Cons

5.2.1 Reliability

Although IPPALE can be a very useful programming aid, IPPALE, as it is currently implemented, is not as reliable as desired. There are two main reasons for this. First, IPPALE is new and does not have all of the bugs worked out. More importantly, however, IPPALE has several problems because it is based on code for an assembler. An assembler is supposed to produce correct output code for legal input code. For illegal input code, however, the assembler only needs to determine and state that the code is illegal. This means that an instruction can be in the middle of being analyzed. Suddenly it is determined to be illegal. The assembler wants to exit immediately ignoring data structures which have been partially created. These half completed instructions cause errors in IPPALE.

Improving the reliability of IPPALE would be rather difficult because it involves serious modifications to the assembler code. This code would have to provide a standard amount of information for any instruction, legal or not. Although difficult, this is not impossible, and would be necessary for a useful tool.

5.2.2 Speed

In this version of IPPALE, the goal was to demonstrate the utility of a real-time instruction analysis and feedback tool. In order to accomplish this some sacrifices were made in the speed of IPPALE in order to make it easily modifiable. Although the speed of the current system is not intolerable, IPPALE would be greatly improved if the speed were increased.

[1] The actual instruction which caused the bug was a LAU operation using a non-D register as the source (store) or destination (load), in combination with a register-to-register move.

The two worst cases are the loading time and the latency between sending an instruction and displaying the relevant information about it.

5.2.3 Adjustable Display Interface

Although the means of displaying information was carefully chosen, another useful improvement to IPPALE would be to allow the programmer to modify the display [11]. Allowing the user to specify the sizes and locations of the blocks in addition to the colors for representation would useful to many people. Novice programmers, for example, may not want to see the instruction block. This representation can be confusing and difficult to understand initially. Similarly, experienced users may choose to use a much smaller message block.

5.2.4 Improved Editor Interface

The interface to the editor could also be greatly improved. Currently instructions must be explicitly sent to IPPALE by the programmer. IPPALE could be modified to send instructions to the Engine Module at appropriate times. For example, IPPALE may send the current instruction every time "Return" is pressed.

5.2.5 Message Library

Currently the only messages displayed are those which are produced by the *ppasm*. These messages do not include warnings about inter-instruction interactions. These types of warnings could be easily added and would be of use for novice programmers. Other information, such as textual explanations for errors would also be extremely useful.

5.2.6 Integration

The design of IPPALE would be greatly improved by improving the integration between IPPALE and *ppasm*. Ideally, the two applications could be developed simultaneously. Developing the two applications simultaneously could greatly reduce the amount of overhead and the latency of IPPALE, especially on start-up.

Another useful addition to IPPALE would be to integrate an on-line help system with IPPALE. This would give the programmer complete access to information about the PP in

one location.

Chapter 6

Conclusions

6.1 IPPALE as an Educational Tool

The current version of IPPALE has already demonstrated the fact that a tool like IPPALE could be extremely useful as an educational tool. This could be used for both educating students on the use and organization of hardware within current microprocessors as well as be used to quickly bring computer programmers up-to-date with new complex microprocessors. A tool like IPPALE allows an interactive method of learning rather than the standard method of reading about microprocessors which offers no feedback to the student.

6.2 IPPALE for Other Processors

Although IPPALE was implemented specifically for the PP of the C8x, a tool like IPPALE would be useful for many different types of microprocessors. The main benefit of IPPALE is in providing information about how the hardware is affected by a single instruction. Therefore, this type of tool would be most useful for a processor which is very complex. An IPPALE-like tool for a RISC processor would probably not be very enlightening.

The use of an IPPALE-like tool for many different processors would also make learning to program each processor much easier. One could imagine, for example, that every Texas Instruments processor or DSP is programmed using an algebraic assembly language similar to the one used for the PP. If each processor also had an IPPALE-like tool for programming, learning to program a new processor would be incredibly easy. The programmer would

already know the syntax rules of the programming language. The usage rules and capabilities of the processor would be easy to understand quickly using the IPPALE-like tool. This would enable microprocessor manufacturers to produce new microprocessors without the worry of training time for future programmers. This would also remove much of the training burden from the manufacturer.

Another advantage for this type of system is that creating code for a new processor would be easier than starting from scratch. Since each of the processors would have the same syntax rules, many of the instructions from old code would not need to be changed. The programmer could simply load the old code into the IPPALE-like tool for the new processor. By stepping through each of the instructions, the programmer could quickly find and fix illegal instructions. This is much more time efficient than starting over from scratch.

6.3 IPPALE for Other Programming Languages

Although IPPALE was implemented for assembly language programming, there are other possibilities for languages for which a tool like IPPALE could be implemented. The basic idea of providing the programmer with information about hardware usage during a specific instruction is most easily associated with low-level programming languages, however, it could also be useful for high-level programming languages.

There are many different ways in which a tool like IPPALE could be used to aid programmers with programming high-level languages. One of these applications would be to create software which displays, on demand, one or more possible compiled versions of part of the high-level code written. Although the final compiled version may be different from the one display, this would still give programmers some feedback and allow programmers to try to create more efficient code as it is being created.

Another possible use would be similar to the previous application, however, rather than display the compiled code purely in text format, the application could have a display similar to IPPALE's. The programmer could then step through the sequence of machine-level instructions and determine which resources are available. This would give the programmer some insight which would make it possible for the programmer to modify the high-level code to create more efficient compiled code.

Appendix A

Description of PP Instructions

A.1 Instruction Classes

The four main functional units of the PP were described in Section 2.2. An algebraic assembly language instruction can have up to four parallel parts: two for the DU, one for the LAU, and one for the GAU. The PFC's operations are all taken care of automatically. These types of instructions are further broken down into what are referred to as major instruction classes. The major classes and an example of each are listed in Table A.1.

```
Miscellaneous (Int Enable/Disable, NOP)  EINT ;Enable all interrupts
6 Operand Operation (MUL, EALU, divi)    d2 = d3 * d4 || d5 = d6 + d7
ALU operation with a 32-bit immediate    d2 = d3 + (0xa52b062a | d1)
ALU operation with a 5-bit immediate     d2 = d3 + (30 | d1)
ALU operation using only register inputs d2 = d3 + (d2 | d1)
Conditional Operation                    d1 =[z] d2 + d3 ;Compute if zero
Non-D register ALU Operation             x2 = d3 + (d4 \& d0)
Local Transfer                           d1 = *(a1+x1)
Local Transfer with Long Offset          d1 = *(a1+9)
Global Transfer                          d1 = *(a9+x9)
Global Transfer with Long Offset         d1 = *(a9+9)
Register-to-Register Move                d1 = d2
Field Move                               d1 =h d2
```

Table A.1: Examples of Different Classes of PP Operations.

The first five major instruction classes listed in Table A.1 are each executed by the DU. Only the second instruction class, the 6 Operand Operation, can be used to effectively

compute two parallel operations. The operations that it can perform in parallel are described in the next section, Instruction Subclasses.

The next two instruction classes, Conditional Operation and Non-D register ALU operation, must be used in combination with another instruction class. These instruction classes are used to modify other instruction classes. The Conditional Operation instruction class can be used with any other operation. If the condition being tested for is met the operation is performed, otherwise, it is not performed.[1] The Non-D register ALU operation instruction class is used to modify any ALU operation. It allows registers other than D registers to be used as either the source or destination for the ALU operation [18].

The next two instruction classes, Local Transfer and Local Transfer with Long Offset, are executed by the LAU. The instructions within this class are usually load or store operations, however there is large variation on what can be computed using this an instruction from this instruction class.

The last four instruction classes from Table A.1 show the different possible instructions which can be executed by the GAU. The Global Transfer and Global Transfer with Long Offset instruction classes have exactly the same functionality as the Local Transfer and Local Transfer with Long Offset instruction classes. They use the global port rather than the local port, however, and can therefore load or store using any of four of the major memory blocks. The local operations can only use one major block of memory. The other two instructions are also executed by the GAU, however they do not use the global port.

A.2 Instruction Subclasses

In order to make it easier to remember what operations can be performed in an instruction, the instruction classes which use the DU are further broken down into several subclasses. The Miscellaneous instruction class has three different subclasses: EINT, DINT, and NOP. These are used to enable or disable interrupt handling and to execute a null instruction.

The 6 Operand Operation instruction class has 5 subclasses:

- MUL || ADD

[1]There is one slight exception to this. If a local or global transfer instruction is performed which replaces the base address register with the new calculated address, this address register replacement is *not* conditional, even though the actual load or store is conditional.

- MUL || SADD
- MUL || EALU
- and EALU || ROT

The first three of these subclasses allow a multiply to be executed in parallel with an add, a shifted add, and an extended ALU operation, respectively. The last subclass allows an extended ALU operation to be performed in parallel with a rotate operation. Please refer to the *Parallel Processor User's Guide* [18] for more information on the numerous restriction for this subclass of instructions.

Each of the other three instruction classes which use the DU have a total of eight instruction subclasses. These subclasses determine the data paths which are used to select three inputs to the ALU. Although there are numerous data paths for each input to the ALU, only eight combinations of these are allowed. These combinations are described in the *Parallel Processor User's Guide* [18].

Appendix B

Description of PP Instruction Word

Table B.1 shows the instruction word bit usage for each of the classes of instructions. For the most part, instructions can be executed in parallel whenever there are no conflicts between used bits. It is important to notice that several of the instruction classes can use different bits depending on exactly what instruction is being encoded and what other instruction classes are being executed in parallel. This is most easily seen in the different usage descriptions of the Local and Global Transfers in Table B.1.

PP Instruction Word Bit Usage

CLASS	Fields (bits 63 → 0)
6 Operand	0 · 1 1 · oper · src3 · dst2 · dst · src1 · src4 · src2 · (unused)
32 bit immed.	1 · class · A · ALU Operation · dst · src1 · 1 1 · (unused) · 32 Bit Immediate
5 bit immed.	1 · class · A · ALU Operation · dst · src1 · 0 · imm src2 · (unused)
Register src2	1 · class · A · ALU Operation · dst · src1 · 1 0 · (unused) · src2 · (unused)
Miscellaneous	1 · 0 0 0 · 1 · - 0 - 0 - 0 - 0 · (unused) · 0 · Operation · (unused)
Local(Long Im)	(unused) · Lmode · d · e · size · s · La · 1 · Lm · bank · L · 0 0 · Local Immediate
Local(PP Rel)	(unused) · Lmode · d · e · size · s · La · (unused) · Lm · (unused) · L · (unused) · Lm/X
Local	(unused) · Lmode · d · e · size · s · La · (unused) · L · (unused) · Lm/X
Local	(unused) · Lmode · d · e · size · s · La · (unused) · L · (unused) · Lm/X
Cond(protect)	(unused) · 0 0 · (unused) · cond · c · x · g · N C V Z · (unused)
Cond	(unused) · cond · (unused)
Global(Long Im)	(unused) · 0 0 · Global Immediate · bank · L · Gmode · xg · e · size · s · Ga · 0 · Gm
Global(PP Rel)	(unused) · Gm/X · bank · L · Gmode · xg · e · size · s · Ga · 1 · Gm
Global	(unused) · Gm/X · (unused) · Gbank · L · Gmode · xg · e · size · s · Ga · (unused)
Move	(unused) · 0 · (unused) · dstbank · (unused) · 0 0 0 0 · src · srcbank · dst · (unused)
Field Move	(unused) · 0 · rm · dstbank · (unused) · 0 0 0 1 · src · e · size · D · dst · (unused)
Non-D DU	(unused) · 0 · (unused) · Adsbank · (unused) · 0 0 1 · (unused) · Asbank · (unused)
Non-D DU	(unused) · dstbank · srcbank · (unused)

Note:
g becomes don't care for Conditional Non-D DU operation

Table B.1: Instruction Bit Usage for Different Classes of Instructions.

References

[1] P. J. Brown, editor. *Writing Interactive Complilers and Interpreters.* John Wiley and Sons, Inc., New York, NY, 1979.

[2] Wilfred J. Hansen. User Engineering Principles for Interactive Systems. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 217–231. McGraw-Hill Book Company, New York, NY, 1984.

[3] Martin Helander, editor. *Handbook of Human-Computer Interaction.* North-Holland, Amsterdam, 1988.

[4] Olaf Krusche. ASMEdit. World Wide Web. Sportlerweg 7, 02692 Doberschau, Germany. http://www.inf.tu-dresden.de/~ok3/asmedit.html.

[5] Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual*, Version 19, Revision 2.4 edition, June 1995.

[6] Vijay K. Madisetti. *VLSI Digital Signal Processors, An Introduction to Rapid Prototyping and Design Synthesis.* Butterworth-Heinimann, Boston, MA, 1995.

[7] Phil Moyse et al. TMS320C80(MVP) Design Methodology. *Texas Instruments Technical Journal*, 12(3):40–55, May-June 1995.

[8] Brad A. Myers, editor. *Languages for Developing User Interfaces.* Jones and Bartlett Publishers, Boston, MA, 1992.

[9] Terrence W. Pratt. *Programming Languages, Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1975.

[10] Charles Rich and Howard E. Shrobe. Initial Report on a LISP Programmer's Apprentice. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive*

Programming Environments, pages 443–463. McGraw-Hill Book Company, New York, NY, 1984.

[11] Bernice E. Rogowitz and Lloyd A. Treinish. Using perceptual rules in interactive visualization. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 2179, pages 287–295, Bellingham, WA, 1994. IS&T - Society for Imaging Science and Technology, Society of Photo-Optical Instrumentation.

[12] Erik Sandewall. Programming in an Interactive Environment: The LISP Experience. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments.* McGraw-Hill Book Company, New York, NY, 1984.

[13] Max Schindler. Interactive software tool generates assembly code for VAX-based systems. *Electronic Design*, 32:312, October 1984.

[14] Axel T. Schreiner. *Using C with curses, lex and yacc, building a window shell for UNIX system V.* Prentice Hall, Great Britain, 1990.

[15] Richard M. Stallman. EMACS: The Extensible, Customizable, Self-Documenting Display Editor. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 300–325. McGraw-Hill Book Company, New York, NY, 1984.

[16] Texas Instruments. *TMS320C80 Multimedia Video Processor (MVP) Technical Brief*, 1994.

[17] Texas Instruments. *TMS320C80 (MVP) Design Seminar Workbook*, 1994. Reference material for 5 day course on programming the TMS320C80.

[18] Texas Instruments. *TMS320C80 (MVP) Parallel Processor User's Guide*, 1995.

[19] Guido van Rossum. An Introduction to Python for UNIX/C Programmers. Technical report, CWI, Amsterdam, January 1993.

[20] Guido van Rossum. *Python Library Reference Manual.* Dept. AA, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, 1.2 edition, April 1995.

[21] Guido van Rossum. *Python Reference Manual.* Dept. AA, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, 1.2 edition, April 1995.