MiniAT 0x1

An Embedded Pedagogical Architecture and Simulator Design

Except where reference is made to the work of others, the work described in this thesis is our own or was done in collaboration with our advisory committee. Further, the content of this thesis is truthful in regards to our own work and the portrayal of others' work. This thesis does not include proprietary or classified information.

William Roberts

Kirk Swidowski

Certificate of Approval:

Saumendra Sengupta
Professor
Department of Computer Science

Mohamed Rezk
Associate Professor
Department of Electrical Engineering Technology

Ronald Sarner
Distinguished Service Professor and Department Chair
Department of Computer Science

MiniAT 0x1

An Embedded Pedagogical Architecture and Simulator Design

William Roberts

Kirk Swidowski

A Thesis

Submitted to the
Graduate Faculty of the
State University of New York Institute of Technology
in Partial Fulfillment of the
Requirements for the
Degree of

Master of Science

Utica, New York
December 16, 2011

MiniAT 0x1

An Embedded Pedagogical Architecture and Simulator Design

William Roberts

Kirk Swidowski

William Roberts

Kirk Swidowski

December 16, 2011

Vita

William Roberts graduated from high school in 2002. He then joined the United States Marine Corps and rose to the rank of Sergeant. He served in support of operation Iraqi Freedom from 2002-2006. After leaving the Marines, William attended Finger Lakes Community College where he was awarded an A.S. in Computer Science in May 2008; he graduated with honors. William then enrolled at the State University of New York Institute of Technology in the accelerated B.S./M.S. in Computer and Information Science. While at SUNYIT, William received a B.S. in Computer and Information Science, Magna Cum Laude, in 2010. William was also awarded the outstanding student award B.S./M.S. Computer and Information Science in 2010, as well as appearing on the presidents list numerous times.

Kirk Swidowski graduated from high school in 2002. He was awarded a CompTIA A+ certification in 2003. During the course of his education Kirk received numerous dean and president list awards. He attended Cayuga Community College and was awarded a A.A.S in Computer Information Systems in 2007. His education continued at the State University of New York Institute of Technology were he received a B.S. in Computer and Information Science, Cum Laude, in 2010. While working full time and attending college full time, Kirk wrote numourus mobile applications, one by the name of TunIt Radio that was published in Smartphone & PocketPC Magazine under an article called "A Windows Mobile Wish List".

William and Kirk presented research at CCSCNE (Consortium for Computing Sciences in Colleges North East) in 2009 at SUNY Pittsburgh.

The authors are currently employed at CACI; in Rome, NY. Kirk as a Software Engineer and William is a Embedded Systems Engineer.

Thesis Abstract

MiniAT 0x1

An Embedded Pedagogical Architecture and Simulator Design

William Roberts

Kirk Swidowski

Master of Science, December 16, 2011
(B.S., State University of New York Institute of Technology, 2010)

158 Typed Pages

Directed by Saumendra Sengupta

The MiniAT 0x1 is a RISC (See Appendix) based embedded architecture for the classroom. It blends contemporary RISC architectures and classroom simulators into one complete unit, that is nothing more than an architecture. The MiniAT features a 4 stage pipeline, prioritized vector interrupts, and a novel ISA format coined the "Very Similar Instruction Architecture"(VSIA). The VSIA format consists of a fixed length instruction word with a single instruction encoding.

The simulator is bundled as a 'C' library that interfaces through a "pins" interface. This generic approach lets the MiniAT adapt to a wide variety of user requirements; thus the MiniAT is designed with a goal that a common simulator platform can be used throughout a CS curriculum, and not just confined to a machine organization or computer architecture class. These classes have traditionally been the territory of pedagogical architectural simulators.

Acknowledgments

The current work (of this thesis) started with Dr. William “Amos” Confer, Kirk Swidowski, David Woytowicz, and William Roberts in fall of 2008 at the State University of New York Institute of Technology. Others have also contributed along the way, and some have left (See the list below). The project challenged everyones understanding of computer architecture and, through our toiling, we hope to present a new computer architecture with both a contemporary and novel design. A big thanks goes out to the following contributors along the way:

- Dr. William Confer - inception to current
- William Roberts - inception to current
- Kirk Swidowski - inception to current
- Dr. Heather Dussault - Fall 2008
- David Woytowicz - Fall 2008
- Dan Brown - Spring 2009

Another thanks goes out to all the people that have helped make this project happen, not just those listed as contributors. To the family of the authors, thank you for your support as we ignored you to develop this architecture.

The project also plans to deliver an open source simulator and other resources to the web at `www.miniat.org`. See section 11 for simulator details.

Style manual or journal used Journal of Approximation Theory (together with the style known as “sunyitms”). Bibliography follows van Leunen’s *A Handbook for Scholars*.

Computer software used The document preparation package TEX (specifically LATEX2e) together with the style-file `sunyitms.sty`.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

During the fall semester of 2008 at SUNYIT, Dave Woytowicz and the authors took an introductory computer organization class. During this class, they were introduced to a simulator called AT Robots [1]. The AT Robots simulator was used for teaching both x86 style assembly and machine organization concepts during the class. This led the team to start evaluating other simulators, searching for a *complete* architectural simulator that was just that, and nothing more. After evaluating both commercial architectures/simulators [22] [13] [8] [96] [90] [52] [34],and educational architectures/simulators [103] [75] [50] [31] [49] [86] [69] [77] [17] [4] [82] [35] [43] [15] [20] [27] [2] [33] [56] [11] [23] [62] [73] [85] [97] [68] [6] [5] [66] [72] [65] [51] [79] [93] [39] [71] [87] [95] [30] [53] [58] [63] [18], the authors noticed that free simulators for commercial processors only simulated parts of the architecture, and were typically just the ISA(See Appendix) [104] [96] [13]. The authors also shared the same feelings as Skrien [86], students should focus on architectural concepts not definitive architectures. The authors also noticed that existing classroom simulators shared the same qualities; they generally simulated only parts of the architecture. Some like LARC [18] do not detail a pipeline and include additional functionality, such as operating system level calls. The authors also noticed that the simulators came pre-bundled, typically with a keyboard and console. Some of them were extensible only by altering the source code. This paper introduces the research of the MiniAT project, and the architecture produced by this effort. The architecture produced is called the MiniAT 0x1 and will be referred to simply as the MiniAT.

1.2 Goals

During the search of existing simulators the authors hoped to discover “bare metal”, architectural simulator, that was detailed to the latch level of the pipeline, if it indeed was pipelined. the authors envisioned the architecture to contain many commercial features to make it applicable to the “real world”. the authors also wanted a simulator that could be easily embedded into different environments. With that said, the authors (and others) set out on the MiniAT project with the following goals in mind:

1. Provide a contemporary blended architecture that contained features from many RISC processors.
2. Provide only an architecture (i.e., no pseudo OS layer).
3. Augment the architecture with an *extensible* and *easy* to use simulator packaged as ‘C’ library.

1.2.1 Blended Approach

The MiniAT project is considered a blended approach because it merges the strong points of existing simulators and commercial architectures. The project provides an architecture for the classroom with many contemporary RISC features. The MiniAT contains a modified “canonical” RISC pipeline similar to DLX [103] [34]. It also contains a zero register, similar to SPARC, Itanium, MIPS, and Power PC [22]. The MiniAT features instruction predication similar to ARM [22] and has a program counter queue(PCQ) similar to MIPS [46]. The MiniAT also introduces a unique Very Similar Instruction Set Architecture(VSIA) design, where all instructions share a single encoding and format.

1.2.2 Bare Metal

The MiniAT is only a definition of an architecture, and nothing else. This ‘bare bones’ approach allows others to build off the MiniAT foundation, and emphasizes that the MiniAT is only a microcontroller and nothing else. Their is no operating system defined, nor support for one designed into the MiniAT. All access to peripherals is through GPIO and a unique MiniAT bus system, as designed by the individual placing a MiniAT into an environment.

1.2.3 Simulator

Another goal of the project was not to just make an architecture on paper, but provide a simulator for it as well. Rather than provide a simulator bundled for a specific purpose, the simulator should be seen as a library where its interface is defined to be similar to that of real hardware(a "pins" interface). For instance, the simulator only requires a binary file assembled into the appropriate format. From that point forward, the main application writes data to the pins, reads data from the pins, and provides a clock. By utilizing a generic "hardware" based interface, peripherals can be "wired" up to the simulator to meet a wide range of functions. See section 11 for a complete description of the simulator.

1.2.4 Pedagogy

Since the main purpose of this architecture was pedagogy the authors wanted to ensure the fundamentals of human learning were understood and how to correctly introduce complicated architectural concepts. After all, this is a "for the students, by the students" project. The authors evaluated papers [59] [74] [60] [24] [54] [83] [76] [64] [3] [26] [100] [55] [99] [12] [21] [25] [28] [98] [101] [7] [29] [40] [78] [80] [10] [42] [44] [67] [41] [45] [88] [47] [14] [37] [36] [48] [57] [61] [89] [32] [38] [92] [94] [70] and found great guidance on how to package everything and provide a significant impact in the classroom. At some point the project intends on releasing teaching aids, instructional labs, and other educational material to augment the MiniAT design in the classroom. The design will be getting its first classroom evaluation in the Spring 2011 undergraduate computer architecture course at SUNYIT.

Chapter 2
Related Work

The goals of the MiniAT project helped to shape the research conducted, and conversely the research helped to shape the goals of the MiniAT project. Since one of the goals of the MiniAT project, was to bring together commercial RISC architectural features in a pedagogic architectural and simulator package, the authors explored the domain of commercial RISC ISAs and pedagogical simulators. It is important to note that a computer architecture can exist without a simulator, however a simulator cannot exist without a computer architecture. This statement allowed the MiniAT to not only pull in architectural concepts from the commercial RISC architectures, but also from the pedagogical simulators as well, after all these are architectures too. The MiniAT design was supposed to be about architectural concepts, not specific architectures, therefore the authors tried to integrate features found in many of the architectures evaluated in the review. Many of these architectures share common functionality, such as a register that contains a constant value of zero. The project also wanted to ensure that the simulator design would be extensible, and extend what others have learned about pedagogical simulators; thus the authors evaluated previous pedagogical simulator designs. This was when it was determined that their is a need for a standalone, library based, pedagogical simulator. All of the simulators evaluated come packaged in a system that cannot be readily modified. Even though source is available, for some, it would require a recompile to get a different environment. Some simulators, even let the user build their own computer architecture, but they limit them in peripheral attachments.

2.1 Commercial

2.1.1 ARM

The ARM[22] architecture was developed in 1983 by ARM Holdings. ARM supports 9 addressing modes:

- Immediate offset - Effective address is base register value plus immediate
- Register offset - Effective address is base register value plus register value
- Scaled register offset - Effective address is base register shifted immediate number of times
- Immediate offset pre-indexed - Effective address is base register value plus immediate, the result is stored in the base register
- Register offset pre-indexed - Effective address is base register value plus register value, the result is stored in the base register
- Scaled register pre-indexed - Effective address is base register shifted immediate number of times, the result is stored in the base register
- Immediate offset post-indexed - Effective address is the base register *only.* Base register plus immediate is then stored into the base register.
- Register offset post-indexed - Effective address is the base register *only.* Base register plus register is then stored into the base register.
- Scaled register post-indexed - Effective address is the base register *only.* Base register shifted immediate number of times is stored into the base register.

ARM supports (at least) 4 instruction formats. All instruction formats support instruction predication, where the 4 high order bits are reserved for this purpose. The different predication formats are given below.

Cond	**Mnemonic**	**Description**	**Condition tested**
0000	EQ	Equal	Z = 1
0001	NE	Not equal	Z = 0
0010	CS/NS	Carry set/Unsigned high or same	C = 1
0011	CC/LO	Carry clear/Unsigned low	C = 0
0100	MI	Minus/Negative	N = 1
0101	PL	Plus/Positive or zero	N = 0
0110	VS	Overflow	V = 1
0111	VC	No overflow	V = 0
1000	HI	Unsigned higher	C = 1 AND Z = 0
1001	LS	Unsigned low or same	C = 0 OR Z = 1
1010	GE	Signed greater than or equal	N = V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	Z = 0 AND N = V
1101	LE	Signed less than or equal	Z = 1 OR N != V
1110	AL	Always (unconditional)	–
1111	(NV)	Never (unconditional)	–

Table 2.1: The codes above apply to all instructions and determine whether the instruction is executed as is, or is executed as a *NOP* instruction. Arithmetic flags determine the result of the predication.

ARM has the ability to pre-process either a shift or rotate instruction on one of the input operands before operating on the instruction. ARM has some instructions that are atypical to RISC instruction sets. It contains *LDM* and *STM* instructions which allow for the transferring of multiple data streams in an instruction, SIMD instructions.

2.1.2 Itanium

The Itanium[22] is a VLIW, RISC based, 64 bit load/store architecture. It provides support for instruction predication, register stacks and speculative execution. Its ISA is based on EPIC design and philosophy. A system control register, called user mask register, is used to control memory access alignment and byte ordering. This register also controls user configurable performance monitors. Big endian byte ordering is the default register byte ordering, and memory access is always in little endian byte ordering.

Explicit parallelism provides hardware level support for instruction bundles. Instruction bundles are where the compiler packs three instructions into a grouping, or bundle. Bundled instructions must not contain *any dependencies*. The processor does not spend time evaluating the instruction stream for instruction level parallelism. Instead the compiler extracts it.

Bundles are organized into instruction groups. It executes the bundled instructions in parallel. Bundles are 128 bits wide and are internally organized into three 41 bit instructions, and a 5 bit template. The template field is used t0 map instruction slots to instruction types. Instructions are categorized into 6 different types, as follows:

- Integer ALU
- non-ALU Integer
- Memory
- Floating Point
- Branch
- Extended

All instructions occupy one instruction slot, except for extended instructions, which occupy two instruction slots in the bundle.

The architecture supports 256 registers:

- 128 integer registers - 64 bit width
- 128 floating point registers - 82 bit width

The large number of registers can be used to make procedure calls and returns extremely efficient. Rather then using a memory based stack, the processor can use a faster register based stack. General purpose registers are divided into static and stack registers. All registers contain a NaT bit which determines whether or not the register is in use. This is used for speculative loading. r0 always contains the constant zero along with its NaT bit also being a zero. Writing to r0 results in an illegal operation. Itanium refer to the PC as the IP register.

There are three addressing modes in Itanium. These modes are similar to the Power PC(See Section 2.1.4). They are:

- Register indirect
- Register indirect with 9 bit immediate
- Register indirect with index

Procedure calls in Itanium are similar to the Power PC. Typical procedure calls do not involve the memory stack; instead they use the register stack for passing operands. Registers 0-31 are considered the static registers and registers 32-127 are considered the stack registers. The register stack frame controls the visibility of stacked registers to a procedure. Similar to SPARC, where register window alignment allows for indirect passing of operands, in Itanium the callers output area is aligned with the callers input area through the *ALLOC* instruction.

Itanium contains, at a minimum, at least 11 instruction encodings. Opcodes are also grouped via a major opcode number similar to SPARC(See Section 2.1.5). Itanium has support for 3 sizes of immediate values:

- 8 bit immediate
- 14 bit immediate
- 22 bit immediate

The immediate sign bit is located at bit position 5 in the encoding. Destination registers for instructions must be from register 0-3.

The architecture supports instruction predication. The result of the comparison is placed in predicate register 1 and its compliment in predicate register 2.

Each branch instruction takes a 21 bit signed instruction pointer relative displacement to the target. Itanium has support for branch prediction and supports branch elimination through its predication mechanism. The hinting mechanism for branches requires 2 bits. The bits are specified as follows:

- Static branch not taken
- Static branch taken
- Dynamic branch not taken
- Dynamic branch taken

The hardware and compiler support speculative loads. If the system bus is busy, when a speculative load is to occur, the speculative load is placed in an Advanced Load Address Table (ALAT). Every store instruction is checked against this table. If their is a match, the entry from the ALAT is removed.

2.1.3 MIPS

The MIPS-32[90][22] is a 32 bit, RISC based, load store architecture. Memory is addressable on byte, wyde, and word boundaries. Once data is loaded all operations are performed on the register file at word granularity. The architecture uses 3 instruction formats, which are:

- Jump
- Register
- Immediate

The MIPS architecture does not have special stack support and only contains minimal support for subroutines. Support for subroutines is included with the jump and link instruction (*JLNK*). The *JLNK* instruction jumps to the specified addresses and places the return value in register 31. Similar to other RISC architectures (i.e., Itanium and SPARC), register 0 always contains a 0. Special purpose high and low registers are used on multiply and divide instructions. On multiplication, this prevents overflow by providing 64 bits of storage for the result of the operation, which could exceed 32 bits. In the case of division, the high register is used for the remainder and the low register contains the quotient.

Even though no hardware enforcement exists for register usage, the convention outlined in table 2.2 is often adhered to.

Name	**Number**	**Intended usage**
Zero	0	Constant 0
$at	1	Reserved for assembler
$v0 - $v1	2 - 3	Results of a procedure
$a0 - $a3	4 - 7	Arguments 1 through 4 (Not preserved across call)
$t0 - $t7	8 - 15	Temporary (Not preserved across call)
$s0 - $s7	16 - 23	Saved temporary (Preserved across call)
$t8 - $t9	24 - 25	Temporary (Not preserved across call)
$k0 - $k1	26 - 27	Reserved for OS kernel
$gp	28	Pointer to global area
$sp	29	Stack pointer
$fp	30	Frame pointer or a saved register $s8
$ra	31	Return address

Table 2.2: MIPS Register Conventions

MIPS supports only a single addressing mode, which is displacement plus contents of base register. The displacement is a 16 bit signed value immediate, and any register can be used as the base address.

2.1.4 Power PC

The Power PC[22] is a 64 bit, RISC based, load/store architecture. It is byte addressable and can be dynamically configured to support big or little endian byte ordering. The architecture supports dynamic switching between 64 and 32 bit modes. There are 64 registers: 32 general purpose integer registers and 32 floating point registers. Both sets of registers are 64 bits wide. Power PC also has 5 special purpose registers. The architecture is divided into three structural units, and each structural unit has registers associated with it(both general purpose and/or special). Instructions are also divided among the structural units(See Table 2.3).

Unit	**Associated Register(s)**
Branch	Control register (CR), link register (LR), count register (CTR)
Fixed Point	32 general purpose integer registers and the fixed point exception register (XER)
Floating Point	32 floating point registers and floating point status and control register (FPSCR)

Table 2.3: PPC Structural Units

The LR is used for the return address for branch and link instructions. The CTR is used for incrementing a loop variable.

The Power PC supports 4 addressing modes which are:

- Register indirect + register
- Register indirect + immediate
- Register indirect with register update
- Register indirect with immediate update

With this architecture, 3 registers can be used with both load and store instructions. Register A and B are used to determine the effective address. Register C, is the destination or source, depending on whether the instruction is a load or store. The immediate, for a load and store, must be expressible in 16 bits or less. Loads work on byte, wyde, word, and double word widths. Data loaded can be either signed or unsigned. This is important on data widths that need to be sign extended to fill a register.

All instructions must be aligned on word boundaries, thus the bottom two bits of instruction addresses are ignored.

The unconditional branch format allows for a 24 bit target address. Bit 30 of the branch format determines if the branch is an absolute or relative branch. Bit 31 can be used to change the branch instruction into a call. When this occurs, the machine places the return address into the LR.

No instruction explicitly moves data between registers. The Power PC has support for some SIMD style instructions. The instruction *LBZU* is a special instruction used for loading arrays. If it is used consecutively, it loads arrays as if incrementing through it in a loop. The *U* designates unsigned, and the *B* designates data widths. Other widths are available, such as:

- b - Byte
- h - Half word (wyde)
- w - Word
- d - Double word

In the following example(See Figure 2.1) the effective address is the sum of rA and rB.

```
lbz rD, disp(rA)

lbzu rD, disp(rA)

lbzx rD, rA, rB

lbzux rD, rA, rB
```

Figure 2.1: Examples of different load instructions for the Power PC.

There are also multi-word load and store (SIMD) instructions(See Figure 2.2), such as:

- *lmw* - Load multiword
- *stmw* - Store multiword

```
lmw rD, disp(rA)

stmw rS, disp(rA)
```

Figure 2.2: Examples of two SIMD instructions for the Power PC.

The architecture also supports branch prediction. When the machine is *in* direct addressing mode, it uses 2 bits, known as the branch input bits (BI)(See Figure 2.4).

Bit Values	**Meaning**
00	No Hint
01	Reserved
10	Branch is likely not to be taken
11	Branch is likely to be taken

Table 2.4: PPC Possible Branch Input Encodings

The direct addressing mode also utilizes 5 bits for branch options (BO):

- Decrement CTR; branch if CTR not equal to 0 and the condition is false
- Decrement CTR; branch if CTR equals 0 and condition is false
- Decrement CTR; branch if CTR not equal to 0 and the condition is true
- Decrement CTR; branch if CTR equals 0 and condition is true
- Branch always
- Branch if condition is false, uses the 2 bit BI field
- Branch if condition is true, uses the 2 bit BI field
- Decrement CTR; branch if CTR not equal to 0, uses the 2 bit BI field
- Decrement CTR; branch if CTR equal to 0, uses the 2 bit BI field

When the machine is in indirect addressing mode, and it is a conditional branch, it takes one additional branch argument,s known as the branch hint operand (BH)(See Table 2.5). Examples of the Power PC's control instructions are give below(See Figure 2.3).

Branch Hint Encoding	
Bit Values	**Meaning**
00	The instruction is a return address from procedure
01	Instruction is not a procedure return, the target address is likely to be the same as the target address used the last time the branch was taken
10	reserved
11	Target address is not predictable

Table 2.5: Power PC Branch Hint

```
;Conditional branch
bc BO, BU, target


;Branch Conditional to a link register
bclr BO, BI, BH
```

Figure 2.3: Power PC example of a conditional branch and branch and link instructions

2.1.5 SPARC

SPARC[22] is based on a RISC II design and comes in both 32 and 64 bit formats, with version 9 and above being 64 bit processors. As with other RISC based processors, register 0 always contains a 0 (Similar to MIPS and Itanium). The general purpose registers can be divided into 3 categories:

- in
- local
- out

SPARC supports extensions to its architecture, which means implementations will vary; however, minimally it will contain 64 registers. Register windows are supported and there can be between 3 and 32 of these windows, again these are implementation dependent. The register windows are also organized as a circular buffer. Each distinct register window contains 24 registers. SPARC also contains a programmer accessible NPC register. SPARC can be configured in a little or big endian memory format(Similar to MIPS, ARM, and Itanium).

The SPARC architecture has support for 2 addressing modes:

- Register indirect with index - Uses two registers, same as immediate offset.
- Register indirect with immediate - Uses a register plus immediate, same as register offset.

Instructions are 32 bits long with two opcode fields. Opcodes are broken down into groupings, where bits 30 and 31 indicate the major field grouping. Thus, all arithmetic instructions would have the same 30 and 31 bit values.

Their are at least 6 instruction encodings. Load and store instructions can be performed in both signed and unsigned mode on byte, half-word, and word data sizes. The *SETHI* instruction stores a 22 bit immediate to the upper 22 bits of the destination register. The call instruction is a delayed instruction. The delay slot can be, and typically is, used to move additional arguments. SPARC uses 3 instruction formats for conditional branch instructions. Call instructions have a specific format and can use a 30 bit signed displacement immediate.

2.2 Simulators

2.2.1 DLX

The paper[103] discusses a simulator called DLX view. It simulates the DLX[34] instruction set and pipeline. Their simulator explains the three versions of the DLX pipelines. The simulator covers the three versions of the DLX pipeline, which are:

- Basic
- Scoreboarding
- Tomosulo's Algorithm

They have a complete GUI incorporated into the simulator for visualization of the pipeline (See Figure 2.4).

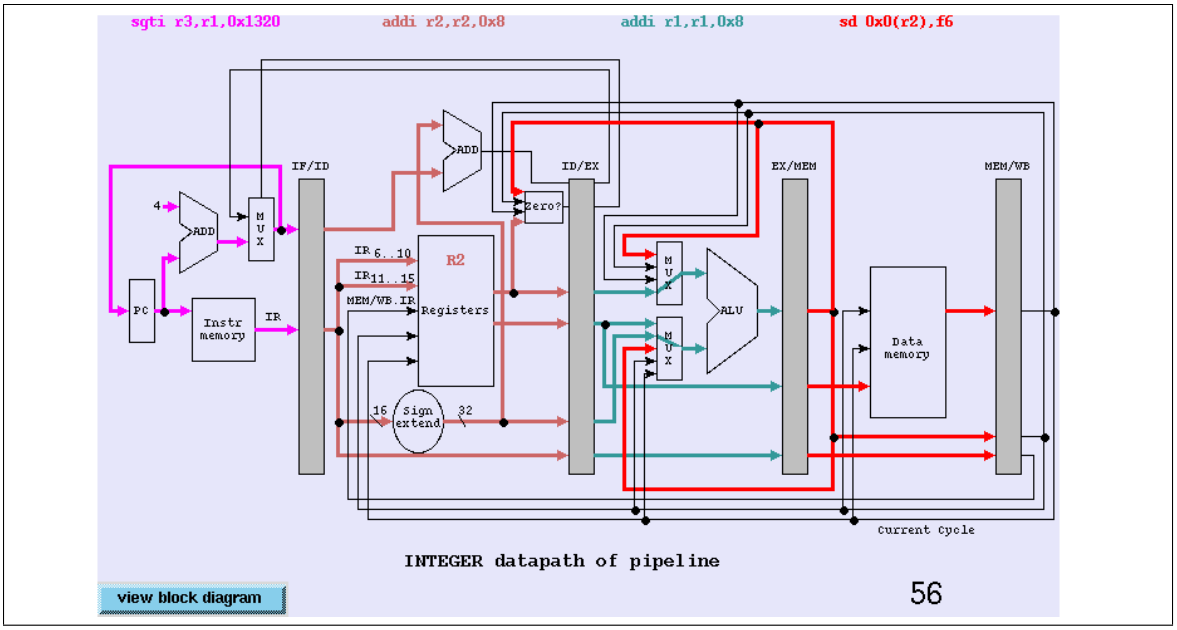


Figure 2.4: A screen shot of the visualization of a DLX pipeline in DLX View. The DLX is a stripped down version of MIPS developed by Hennesey and Paterson[34]. The DLX View snapshot was found at `cobweb.ecn.purdue.edu/~teamaa/dlxview/`

2.2.2 Postroom Computer

The Postroom Computer [73] is an emulator of the Little Man Computer [73] which supports 10 addressing modes, they are:

- Register Direct - Effective address is the register number, operand value is what is in the register.

- Base Direct - Effective address is the value a register added to the value in a base register, operand value is what is at the effective address.
- Immediate Direct - Effective address none, operand value is the immediate.
- Predecrement direct - Effective address is the value in a register decremented by one, operand value is what is at the effective address.
- Postincrement direct - Effective address is the value in a register and is incremented by one afterword, operand value is what is at the effective address.
- Register indirect - Effective address is the value in a register, operand value is what is at the effective address.
- Base indirect - Effective address is the value in a register added to a base register, operand value is what is at the effective address.
- Immediate indirect - Effective address is the immediate, operand value is what is at the effective address.
- Predecrement indirect - Effective address is the value at the address of a value in a register decremented by one, operand value is what is at the effective address.
- Postincrement indirect - Effective address is the value at the address of a value in a register and is incremented by one afterword, operand value is what is at the effective address.

The Postroom computer also supports an interrupt system. The architecture contains an interrupt register(IRP). If this register contains a nonzero value, an interrupt could occur if interrupts are enabled. The *MSK* instruction will determine if an interrupt will occur, by masking the IRP with the interrupt mask register(IMK). If the result is a nonzero, value then an interrupt will be triggered.

2.2.3 ANT-32

The ANT-32[33] was developed at Harvard University as a pedagogical architecture. The ANT-32 is a 32 bit, load/store, RISC based processor. The ANT-32 has support for privileged execution, and thus has support for operating systems. The memory layout provides both word and byte addressing and is laid out in a big endian format. The ANT-32 is a paged architecture with 4k block sizes. Memory is managed through a translation look-a-side buffer based Memory Management Unit(MMU). The MMU provides support for 1 gigabyte of physical memory starting at address 0, and it does not need to be physically contiguous. The ANT-32 has multiple instruction encoding schemes. All instructions are of a fixed 32 bit width and must be aligned on word boundaries. The high byte, of the instruction word, contains the opcode, and subsequent register fields have a width of 8 bits. Immediate values are encoded as either 8 bit or 16 bit values depending on the instruction. The ANT-32 is a 3 address machine.

Although ANT-32 only has one addressing mode, the machine offers a plethora of accessing methods. Since register 0 always returns a 0, utilization of this register will allow for an immediate indirect addressing mode. Using one of the 32 registers and supplying an immediate value of 0, the processor can obtain a register indirect addressing mode. By combining base registers and immediate values, one can provide alternate addressing modes as well.

The architecture also supports both internal and external interrupts. The external interrupt is triggered through a single interrupt request line. Interrupts can be disabled independently of each other.

ANT-32 provides a few unique instructions and are listed below.

- srand - Seeds the random number generator
- rand - Provides a pseudo random number

- cin - Provides console input
- cout - Provides console output

2.2.4 LOGO

LOGO[75], is a robot (See Figure 2.5) designed by MIT for elementary school education. The turtle obeys simple commands, such as:

- Pen up
- Pen down
- Rotate
- Forward X, where X corresponds to how many forward units the robot will move.
- Right Y, where Y is the number of degrees the robot will rotate.

The turtle can be used to draw shapes on paper, but that is not all. It was also used to balance objects on its surface.

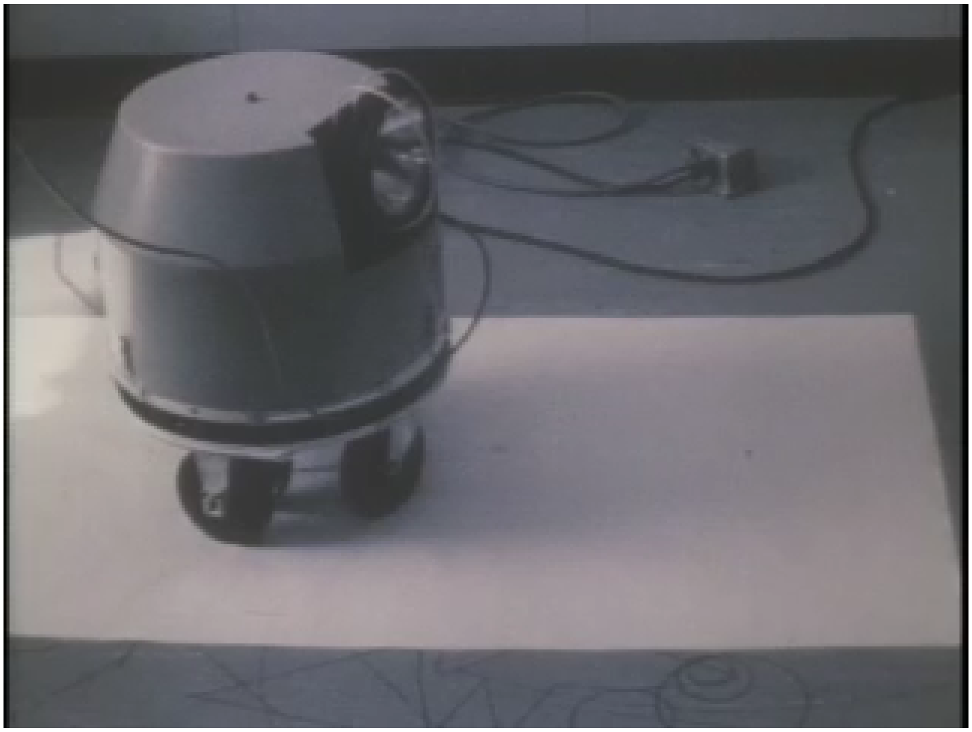


Figure 2.5: An image of a LOGO turtle drawing a picture on a piece of paper. This photo courtesy of `http://csail.mit.edu`

2.2.5 Kerridge Simulator

The Kerridge Simulator[50] is a general purpose architectural simulator, simulated at the register transfer level. By simulating at the register transfer level, it provides an architectural neutral way for hardware components to be integrated together. These components include:

- Registers
- Flip flops
- Slow peripheral channels
- Stores
- Stacks

It was decided not to implement a bus. All components have configurable access times and widths.

The Kerridge simulator is controlled through a set of configurable micro instructions. If the computer is accurately defined, then the output will be printed for every instruction executed along with the initial and final states of the machine. The simulator is written in PASCAL.

2.2.6 Simple Machine

The Simple Machine[49], was designed for introductory computer science courses. It simulates the basics of a computer and is made available with an assembler and loader. The machine uses 16 bit fixed width instructions with a 4 bit opcode. It uses a single accumulator and a single index register. One bit for the instruction is used to indicate indexed operations, leaving the other 11 bits for addressing the 2048 words of memory. The word size is 16 bits. It is a 2's compliment machine. The machine maintains a single function call stack, where the stack pointer is *not* user accessible. The instruction set has been provided for reference(See Figure 2.6).

OP Code	**Mnemonic**	**Explanation**
0000	HALT	End virtual execution
0001	READ	Read 1 character from the keyboard
0010	WRITE	Write 1 character to the screen
0011	LDA	Load accumulator from memory
0100	STA	Store accumulator to memory
0101	JSUB	Jump to sub routine
0110	RSUB	Return from sub routine
0111	ADD	Add operand to accumulator
1000	SUB	Subtract operand from accumulator
1001	JMP	Jump to address
1010	JPOS	If $acc > 0$ then jump to address
1011	JNEG	If $acc < 0$ then jump to address
1100	JZER	If $acc == 0$, then jump to address
1101	ZERX	Set index register equal to 0
1110	INCX	Increment index register
1111	DECX	Decrement index register

Table 2.6: Listing of Simple machine instructions

The Simple machine comes bundled with an assembler and loader. The machine was utilized in undergraduate computer science courses with a focus on how software was built and loaded into memory.

2.2.7 Little Man Computer

One of the first simulators designed to abstract hardware and provide pedagogical value was the Little Man Computer [102]. The Little Man Computer was introduced in 1965 by

Stuart Madnick of MIT. The Little Man Computer has many features that make it ideal for the classroom, such as:

- A small and concise instruction set.
- A simple to use interface.
- The ability to accept both assembly and binary files.

However, the Little Man Computer, henceforth known as LMC, is designed to abstract memory and provide a working model for computer scientists to follow. The LMC helps visualize the idea of memory being thought of as a single dimensional array (to the programmer). The LMC also helps the student understand the concept of a program counter and its contextual use in branches. Some branches are PC relative and require arithmetic to determine the effective branch address. Other branches may be absolute, like a goto that is hard-coded to the beginning of RAM. The LMC does not simulate interrupts, but some models of the LMC allow a syscall instruction that allows one to gain access to hardware through a pseudo operating system environment.

2.2.8 LARC

LARC[18], developed at Hobart and William Smith College by Marc Corliss in 2009, is another RISC type simulator. It has a small instruction set molded after the MIPS ISA . It also comes complete with a Java like compiler and graphical debugger. LARC performs I/O similar to other simulators, by providing a syscall interface. LARC is rather nice as it reduces the MIPS ISA to a few core instructions, reducing the time it takes to master the ISA and start writing programs. It is also well documented, comes with classroom aids, and is easy to use.

2.2.9 ESCAPE

The ESCAPE[15] simulator consists of two different simulators. The instruction set architectures of both machines are essentially identical, even though the micro-architectural aspects are very different (a micro-programmed processor versus a pipelined processor). The instruction set architecture was inspired by Hennessy and Patterson's DLX. The DLX only utilizes 3 encodings and so does ESCAPE. All instructions have a 32-bit encoding. The control unit is micro-coded. The microcode address is kept in a special register, the uAR. Both the control unit and the data path are pipelined into the 5 traditional RISC stages of instruction fetch, instruction decode, execute, memory, and writeback. ESCAPE also comes with a plethora of tools for the classroom, such as:

- Memory viewers
- Assembler/disassembler
- Step by step execution diagrams
- Microcode editor
- The ability to trace program execution
- Active pipeline diagrams

First experiments with ESCAPE[15] revealed significant improvements in teaching effectiveness. Students responded very positively, and the evaluations indicate a far deeper understanding than was previously attainable by using only the traditional textbook-and-paper-problems approach. ESCAPE is one of the more complete simulators available; it has a nice instruction set, comes well packaged, and has few encodings. However, ESCAPE still provides a syscall interface and does not simulate the pipeline in it's entirety, such as the latches and control logic.

2.2.10 OAMulator

In 2001, Menczer introduced the OAMulator[62]. The OAMulator is a one address machine with the following assumptions:

- Infinite memory
- program instructions start at address 0x01
- Memory mapped I/O at address 0x00

The OAMulator (OAM) has 5 registers: the accumulator (ACC), the B register, the PC, the instruction register (IR), and the address register (AR). The OAM can only access the accumulator or one memory location per instruction. Outside of load and store instructions, the OAM ALU instructions support going to memory, therefore arithmetic instructions can use memory. The OAM consists of a high level programming language and a low level programming language, and is useful for teaching a variety of computer science concepts. In fact Menczer recommends their simulator for teaching the following concepts:

- Von Neumann architecture concepts
- Registers
- ALU
- Stages of execution
- Instruction set architecture
- Assembly languages
- memory and addressing
- high-level languages
- compilers

- I/O assignment and control statements

- Variable reference resolutions

- expressions and parsing

- Optimization

The OAMulator is not pipelined and does not support interrupts.

2.2.11 STARTLE and CPU SIM

STARTLE[86] is a simulator based on the register transfer language level, and developed by Skien. The simulator only runs on a Macintosh computer. It does allow students to design their own machines from scratch, similar to the Kerridge Simulator. However, Skrien felt students spent too much time configuring the simulator and not enough time focusing on the simulation itself. This led to a new simulator called CPU SIM[85], which is designed for CS3 level students. CPU SIM allows students to define different parts of the architecture allowing them to achieve hands on experience across a breadth of designs, rather than just a particular architecture. The user specifies these different parts of the architecture in register transfer language. There the user can define parameters such as:

- Number of registers

- Amount of RAM

- Micro instructions

- Machine instructions

- Assembler level instructions

CPU SIM does have some limitations, as pointed out in the paper. CPU SIM intentionally lacks the following architecture topics:

- Floating point number representations

- Computer organization below the microcode level
- Issues concerning speed, such as pipelining, parallel processors and the use of caches
- Interrupts and traps
- Operating system level concepts
- The linking and loading of separate assembly modules

Skrien states that, “Adding features [to] CPU SIM to address these topics would have made it significantly more complex and so, in the author’s view, less pedagogically useful for the intended audience.”

2.3 Others

2.3.1 The European HP PC Queue Patent

During the implementation of the MiniAT, a problem arose when multiple stages were writing to the PC in the same cycle. The problem was solved with a PCQ (See Section 6.2.11). Originally, the authors were under the impression that this was a unique feature, however, an HP patent[46] that illustrates the same concept was found.

The paper discusses that around 10 - 25% of instructions executed are branch instructions. Because of the overhead of branching, by the need to potentially flush the pipeline, processors started using delayed branch instructions to reduce this performance penalty. However, the delayed branch causes new problems. One of the issues that develops is that when interrupts occur soon after the delayed branch instruction is fully executed, the PC would only contain the branch target and the delay slot instruction(s) would be lost.

The issue is resolved by introducing a PCQ. By allowing all the stages to push their program counters to the PC queue, all instructions (including the delay slot), will execute on return from the interrupt. This is accomplished taking the PC for every stage in the pipeline and backing it up to the PCQ. The stage PCs are backed up in reverse order, starting from the writeback stage and ending with the fetch stage. After that, the actual PC is placed on the queue. At this point, the head of the queue contains the writeback instruction and the tail contains the target. This is so when the PCQ is restored, on returning from the interrupt, execution will start in the correct order. In summary a PCQ is needed if an interrupt occurs when the pipeline includes partially executed instructions corresponding to non sequential memory addresses in the program.

According to this patent, MIPS introduced the PCQ. The size of the PCQ must be $n-1$, where n is the number of pipeline stages.

Chapter 3

Overview

The MiniAT, or MiniAT 0x1, is a 32-bit pipelined RISC based microcontroller architecture. A modified Harvard design is utilized with little endian byte ordering and a word size of 32 bits. The MiniAT uses a *single instruction encoding.* The MiniAT has support for 4 groups of instructions, which are:

1. Arithmetic
2. Logic
3. Memory
4. Conditional

All instructions can be predicated and are of a double word length, which allows for full word immediate values. The conditional instructions are of an atomic test and branch format that supports hinting for branch prediction. The architecture adheres to a load/store format that includes support for a register-based stack.

The architecture provides for many integrated resources, such as:

- 256 word size registers
- 32 kilobytes of flash memory
- 24 kilobytes of Random Access Memory (RAM)
- Four 16 bit General Purpose Input and Output (GPIO) ports
- External asynchronous return to 0 bus

- 27 prioritized interrupts
- Clock pre-scaled watchdog timer
- 4 general purpose timers

Chapter 4
Layout

4.1 Overview

In computing, memory is organized in a logical single dimensional array. Memory addresses can correspond to physically separate memory devices, peripherals, and incur special meaning dependent to the architecture. For instance, writing a 5 to memory address 0x1234 may put a 5 in a RAM chip mapped to that address. If the processor is using memory mapped I/O and that address corresponds to that, then a peripheral on the bus would have received a 5. It could have also wrote a 5 to a memory mapped register that controls functionality of the machine, such as unsigned vs signed operation. Typically memory is organized into a hierarchy, with registers being the fastest and hard disks, or permanent storage, being the slowest (See Figure 4.1).

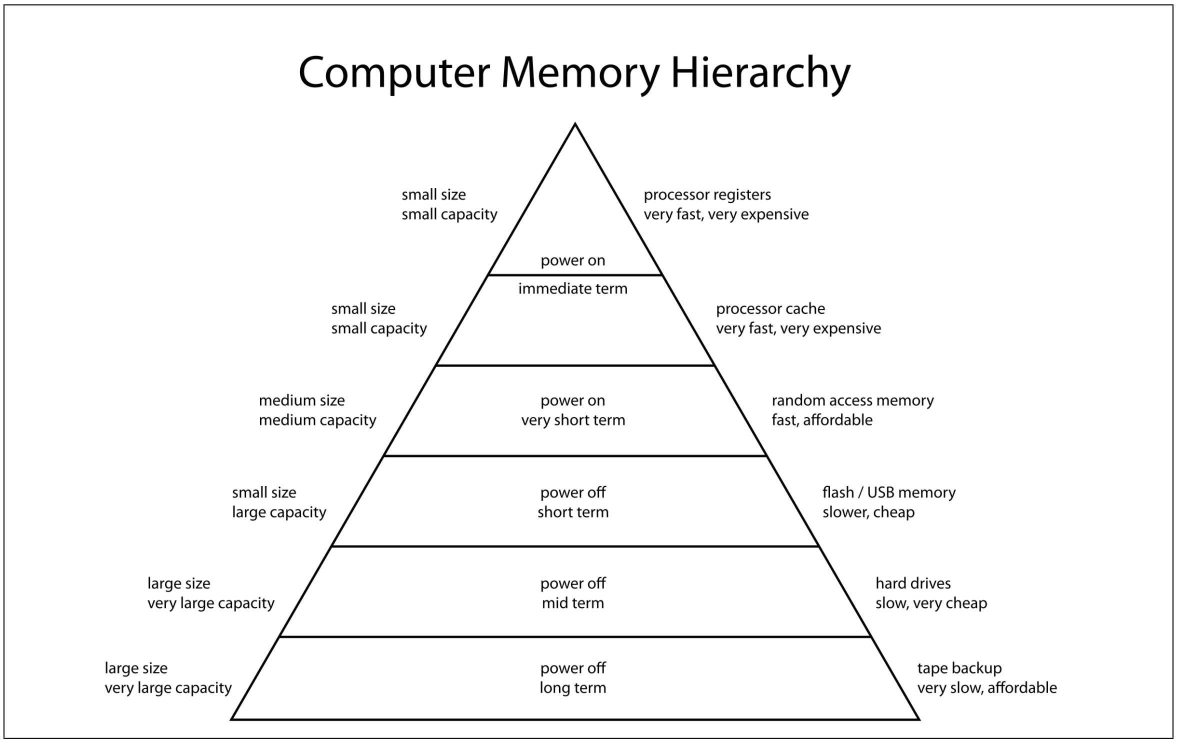


Figure 4.1: The canonical memory hierarchy. Picture courtesy of `www.wikipedia.org`

4.2 MiniAT

4.2.1 Memory

Memory is organized into word addressable blocks starting at address 0x0000 to 0xFFFF, thus allowing for a size of 256 kilobytes. Instructions are guaranteed to be aligned on word boundaries because the microcontroller is *only* word addressable. On boot, instruction execution starts at the beginning of flash. Code is relocatable from flash to RAM.

4.2.2 Memory Map

0x0000 **...** **0x01FF**	**Reserved**
0x0200 **...** **0x19FF**	**RAM**
0x1A00 **...** **0x1AD9**	**Reserved**
0x1ADA **...** **0x1B19**	**Interrupt Vectors** **0x1ADA** INT0: Reset **0x1ADB** INT1: Illegal Address **0x1ADC** INT2: Illegal Instruction **0x1ADD** INT3: Bus Error **0x1ADE** INT4: Divide by Zero **0x1ADF** INT5: Watchdog **0x1AE0** INT6: Timer 0 **0x1AE1** INT7: Timer 1 **0x1AE2** INT8: Timer 2 **0x1AE3** INT9: Timer 3 **0x1AE4** INT10: Overflow **0x1AE5-0x1AF9** Reserved

Table 4.1: Memory Map

0x1ADA **...** **0x1B19**	**Interrupt Vectors (Continued)** **0x1AFA** INT32: External Interrupt 0 **0x1AFB** INT33: External Interrupt 1 **0x1AFC** INT34: External Interrupt 2 **0x1AFD** INT35: External Interrupt 3 **0x1AFE** INT36: External Interrupt 4 **0x1AFF** INT37: External Interrupt 5 **0x1B00** INT38: External Interrupt 6 **0x1B01** INT39: External Interrupt 7 **0x1B02** INT40: External Interrupt 8 **0x1B03** INT41: External Interrupt 9 **0x1B04** INT42: External Interrupt 10 **0x1B05** INT43: External Interrupt 11 **0x1B06** INT44: External Interrupt 12 **0x1B07** INT45: External Interrupt 13 **0x1B08** INT46: External Interrupt 14 **0x1B09** INT47 External Interrupt 15 **0x1B0A-0x1B19** Reserved

Table 4.2: Memory Map Continued

0x1B1A	**General Purpose I/O Pins** **0x1B1A** Port A **0x1B1B** Port B **0x1B1C** Port C **0x1B1D** Port D
0x1B1E	**System Control Registers** **0x1B1E** System Register **0x1B1F** Watchdog Compare **0x1B20** Watchdog Count **0x1B21** Timer Control **0x1B22** Timer 0 Compare **0x1B23** Timer 1 Compare **0x1B24** Timer 2 Compare **0x1B25** Timer 3 Compare **0x1B26** Timer 0 Count **0x1B27** Timer 1 Count **0x1B28** Timer 2 Count **0x1B29** Timer 3 Count **0x1B2A** Timer I/O pins **0x1B2B..0x1F15** Reserved

Table 4.3: Memory Map Continued

	0x1F16 Port A Directions **0x1F17** Port B Directions **0x1F18** Port C Directions **0x1F19** Port D Directions **0x1F1A** IERA Interrupt Enable Register A **0x1F1B** IERB Interrupt Enable Register B **0x1F1C** IFFA Interrupt Flag Register A **0x1F1D** IFRB Interrupt Flag Register B
0x1F1E **...** **0x1FFF**	**Virtual Machine Identification (ROM)**
0x2000 **...** **0x3FFF**	**Flash**
0x4000 **...** **0xFFFF**	**Peripheral Memory Space**

Table 4.4: Memory Map Continued

4.2.3 Registers

The MiniAT has 256 registers which are 32 bits wide. Only two registers hold a fixed special purpose with the rest being general purpose. Registers 1, 253 and 254 can hold a special purpose based on the instruction or agreed upon as general usage. The table 4.5 and 4.6define the purpose and usage of these special registers.

Register	**Usage**
r0	Zero register always contains a 32 bit zero value. This register is read-only, but it may be used as the destination register for any instruction. The zero register is not directly subject to data hazards, nor can it create them. i.e., the instruction `ADD r0, r2, r0, 3  ;r0 = r2 + 3` will not stall any future instruction that uses r0 as an input, nor will it be stalled by a previous instruction writing to r0. However, this instruction may be stalled by previous instructions writing to r2.
r1	The predicate register is used by a predicating instruction (instructions with the P bit of the encoding set) to determine whether it should execute or be treated as a *NOP*. If r1 contains a non-zero value, the instruction is treated as a *NOP*.
r253(RSP)	The Register Stack Pointer (RSP) is suggested for use when synthesizing a stack within the register file using the RLOAD and RSTORE instructions. These two instructions work identically to *LOAD* and *STORE* except the displaced register calculation provides the index into the register file, whether source or destination, rather than an address in memory. It's purpose is not to be confused with the register stack frame pointers or register frame pointers of other architectures (e.g., Itanium) where register stacking is specifically for parameter passing, calls, and returns. MiniAT 0x1 register stacks are not nearly so limited.
r254(SP)	The Stack Pointer (SP) register is suggested for use when synthesizing a stack in memory using the *LOAD* and *STORE* instructions.

Table 4.5: MiniAT Suggested Register Usage

Register	**Usage**
r255(PC)	The Program Counter (PC) register contains the address of the next instruction and is the primary latch into the fetch stage of the pipeline. It is not in the register file, although it is indexed as such. The PC is not read-only but is otherwise similar to r0; it may be used as the destination register for any instruction, and it is not directly subject to data hazards, nor can it create them. This allows branches with flexible-length delay slots to be achieved using any ALU instruction, $LOAD$, and $RLOAD$ because the PC is *not* affected until the writeback stage of the pipeline. On the other hand, conditional branch instructions employ a branch predictor that modifies the PC, should the prediction indicate a taken branch, by the end of the decode stage of the pipeline.

Table 4.6: MiniAT Suggested Register Usage Continued

Chapter 5
Instruction Set

5.1 Overview

The instruction set is the set of instructions that are supported by a particular machine. For instance, a machine may have multiply, addition and subtraction instructions. These instructions would be part of the instruction set. Instruction sets are unique to machines; therefore, different machines may have different ISAs. These instructions contain data for the machine to act on, and this data is formatted in a unique manner. This format is known as an encoding. Machines may have numerous encodings dependent on the instruction being encoded. ISAs can also support features like instruction predication[22]. This is a feature where conditional execution of instructions can occur. ISAs can also support branch hinting[52][22][34]. This mechanism provides a hint to the branch prediction mechanism as to whether or not the branch is likely to be taken.

5.2 MiniAT

The MiniAT utilizes a new ISA format called the Very Similar Instruction Architecture, or VSIA for short. What makes instructions similar is that *all* instructions utilize a single encoding. Also, most instructions follow a rA = rB operator (rC + 32 bit immediate) format. All instructions can be predicated and branches can be hinted.

The original ISA design for the MiniAT included 14 encodings and 128 instructions. Through the use of the rA = rB operator (rC + immediate) formatting, the instruction set was reduced to 22 instructions while increasing functionality. In the original ISA design, it was not possible, in a single instruction, to subtract a register from an immediate and store the result in a destination register. The new ISA allows for this(See Figure 5.1).

```
sub r5, r0, r6, -3
```

Figure 5.1: Where the ISA declarations are defined as: rA = r5, rB = r0 (Always contains the constant 0), rC = r6, Immediate = -3

It is important to note that rC + I takes precedence over the defined operator. The above statement is equivalent to r5 = 0 - (r6 +- 3), which reduces to r5 = -r6 + 3. This, again, can be simplified through the commutative property to be r5 = 3 - r6.

The use of a full 32 bit immediate also allows a 32 bit number to be moved into a register in a single instruction. By doing this, MiniAT assembly programmers avoid having to write code that takes multiple instructions to load a full word immediate into a register(See Figure 5.2 and 5.3).

```
;Loading mask
;0xDEADBEEF = 1101111010101101_1011111011101111
;into rX in MIPS
addi,$rX,1101111010101101b
sll $rX,16
or,$rX,1011111011101111b
```

Figure 5.2: MIPS - Moving 32Bit Immediate Into a Register

Instead MiniAT assembly writers write this:

```
;Loading mask
;0xDEADBEEF = 1101111010101101_1011111011101111
;into rX in MiniAT
add rX,0xDEADBEEF
```

Figure 5.3: MiniAT - Moving 32Bit Immediate Into a Register

5.2.1 Addressing Modes

The MiniAT supports only a single addressing mode. This addressing mode is very similar to MIPS (See section 2.1.3). The key differences is that the MiniAT supports a full 32 bit (word) immediate. Also, the immediate can be either signed or unsigned and depends on the mode of the machine.

5.2.2 Encoding

As stated above, all instructions utilize the same encoding.

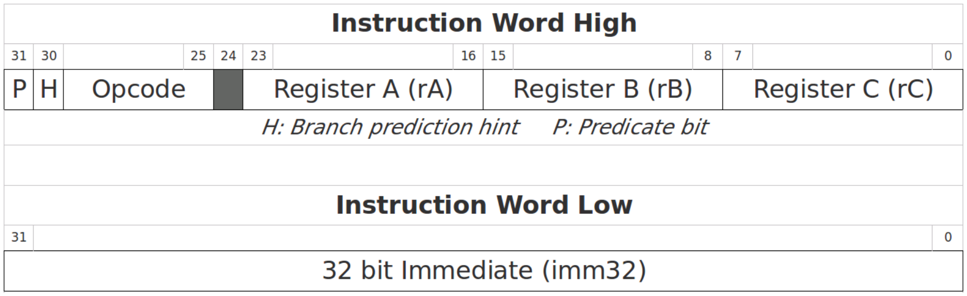


Figure 5.4: MiniAT Instruction Encoding Format

5.2.3 Instruction Groups

All instructions fall into one of 4 categories, which are:

- Arithmetic (See section 5.2.4)
- Logical (See section 5.2.5)
- Memory Access (See section 5.2.6)
- Conditional (See section 5.2.7)

5.2.4 Arithmetic Instructions

ADD - `Addition`

`General usage:`

ADD rA, rB, (rC + imm32)

Operation Detail:

reg (rA) = reg (rB) + (reg (rC) + imm32);

Opcode:

0x00

Example Usage:

ADD r6, r4, (104 + r5)

Example Encoding (bin):

00000 ??P 00000110 00000100 00000101 00000000_00000000_00000000_01101000

Example Encoding (hex):

0x0006040500000068

SUB - Subtraction

General usage:

SUB rA, rB, (rC + imm32)

Operation Detail:

reg (rA) = reg (rB) - (reg (rC) + imm32);

Opcode:

0x01

Example Usage:

SUB r3, r8, (15 + r5)

Example Encoding (bin):

00001 ??P 00000011 00001000 00000101 00000000_00000000_00000000_00001111

Example Encoding (hex):

0x080308050000000F

MULT - Multiplication

General usage:

MULT rA, rB, (rC + imm32)

Operation Detail:

reg (rA) = reg (rB) * (reg (rC) + imm32);

Opcode:

0x02

Example Usage:

MULT r13, r7, (30 + r2)

Example Encoding (bin):

00010 ??P 00001101 00000111 00000010 00000000_00000000_00000000_00011110

Example Encoding (hex):

0x100D07020000001E

DIV - Division to find quotient

General usage:

DIV rA, rB, (rC + imm32)

Operation Detail:

reg (rA) = reg (rB) / (reg (rC) + imm32);

Opcode:

0x03

Example Usage:

DIV r8, r5, (14 + r7)

Example Encoding (bin):

00011 ??P 00001000 00000101 00000111 00000000_00000000_00000000_00001110

Example Encoding (hex):

0x180805070000000E

MOD - Modulus division to find remainder

General usage:

MOD rA, rB, (rC + imm32)

Operation Detail:

reg (rA) = reg (rB) % (reg (rC) + imm32);

Opcode:

0x04

Example Usage:

MOD r10, r20, (17 + r30)

Example Encoding (bin):

00100 ??P 00001010 00010100 00011110 00000000_00000000_00000000_00010001

Example Encoding (hex):

0x200A141E00000011

5.2.5 Logical Instructions

AND - Bitwise AND

General usage:

AND rA, rB, (rC + imm32)

Operation Detail:

reg (rA) = reg (rB) & (reg (rC) + imm32);

Opcode:

0x09

Example Usage:

AND r2, r3, (0x98765432 + r4)

Example Encoding (bin):

01001 ??P 00000010 00000011 00000100 10011000_01110110_01010100_00110010

Example Encoding (hex):

0x4802030498765432

OR - Bitwise OR

General usage:

OR rA, rB, (rC + imm32)

Operation Detail:

reg (rA) = reg (rB) | (reg (rC) + imm32);

Opcode:

0x0A

Example Usage:

```
OR r5, r6, (0xBADCC0DE + r7)
```

Example Encoding (bin):

```
01010 ??P 00000101 00000110 00000111 10111010_11011100_11000000_11001110
```

Example Encoding (hex):

```
0x50050607BADCC0DE
```

EXOR - Bitwise Exclusive OR

General usage:

```
EXOR rA, rB, (rC + imm32)
```

Operation Detail:

```
reg (rA) = reg (rB) ^ (reg (rC) + imm32);
```

Opcode:

```
0x0B
```

Example Usage:

```
EXOR r8, r9, (r10 + 0xFEEDCAFE)
```

Example Encoding (bin):

```
01011 ??P 00001000 00001001 00001010 11111110_11101101_11001010_11111110
```

Example Encoding (hex):

```
0x5808090AFEEDCAFE
```

SHL - Shift left

General usage:

```
SHL rA, rB, (rC + imm32)
```

Description:

Logically shifts bits to the left a calculated number of positions, bringing 0 into opened low order bits.

Operation Detail:

```
reg (rA) = reg (rB) << (reg (rC) + imm32);
```

Opcode:

```
0x0C
```

Example Usage:

SHL r11, r12, (r13 + 0xBA5EBA11)

Example Encoding (bin):

01100 ??P 00001011 00001100 00001101 10111010_01011110_10111010_00010001

Example Encoding (hex):

0x600B0C0DBA5EBA11

SHR - Shift right

General usage:

SHR rA, rB, (rC + imm32)

Description:

Logically shifts bits to the right a calculated number of positions, bringing 0 into opened high order bits.

Operation Detail:

reg (rA) = reg (rB) >> (reg (rC) + imm32);

Opcode:

0x0D

Example Usage:

SHR r14, r15, (r1 + 0xB01DFACE)

Example Encoding (bin):

01101 ??P 00001110 00001111 00000001 10110000_00011101_11111010_11001110

Example Encoding (hex):

0x680E0F01B01DFACE

5.2.6 Memory Access Instructions

LOAD - Load a register from memory

General usage:

LOAD rA, [rC + imm32]

Description:

Register A is loaded with a copy of the data at the effective address found by adding the contents of register C to an immediate.

Operation Detail:

reg (rA) = mem (reg (rC) + imm32);

Opcode:

0x10

Example Usage:

LOAD r2, [r3 + 4]

Example Encoding (bin):

10000 ??P 00000010 ???????? 00000011 00000000_00000000_00000000_00000100

Example Encoding (hex):

0x8002000300000004

STORE - Store a register value in memory

General usage:

STORE [rC + imm32], rA

Description:

The data in register A is copied to the effective address found by adding the contents of register C to an immediate.

Operation Detail:

mem (reg (rC) + imm32) = reg (rA);

Opcode:

0x11

Example Usage:

STORE [r4 -1], r5

Example Encoding (bin):

10001 ??P 00000101 ???????? 00000100 11111111_11111111_11111111_11111111

Example Encoding (hex):

0x88050004FFFFFFFF

RLOAD - Load a register from an indexed register

General usage:

RLOAD rA, [rC + imm32]

Description:

Register A is loaded with a copy of the data in the register with the effective index found by adding the contents of register C to an immediate.

Operation Detail:

```
reg (rA) = reg (reg (rC) + imm32);
```

Opcode:

0x12

Example Usage:

```
RLOAD r2, [r3 + 4]
```

Example Encoding (bin):

```
10010 ??P 00000010 ???????? 00000011 00000000_00000000_00000000_00000100
```

Example Encoding (hex):

```
0x9002000300000004
```

RSTORE - Store a register value in an indexed register

General usage:

RSTORE [rC + imm32], rA

Description:

The data in register A is copied to the register with the effective index found by adding the contents of register C to an immediate.

Operation Detail:

```
reg (reg (rC) + imm32) = reg (rA);
```

Opcode:

0x13

Example Usage:

```
RSTORE [r4 -1], r5
```

Example Encoding (bin):

```
10011 ??P 00000101 ???????? 00000100 11111111_11111111_11111111_11111111
```

Example Encoding (hex):

0x98050004FFFFFFFF

5.2.7 Conditional

BRAE - Branch if equal

General usage:

BRAE hint [rC + imm32], rA, rB

Description:

The Program Counter (PC) register is replaced by the effective address found by adding the contents of register C to an immediate only if the value in register A is equal to the value in register B.

Operation Detail:

```
if (reg (rA) == reg (rB))
   reg (PC) = reg (rC) + imm32;
```

Opcode:

0x16

Example Usage:

BRAE [r2 + 0x1ADA], r3, r4

Example Encoding (bin):

10110 ?HP 00000011 00000100 00000010 00000000_00000000_00011010_11011010

Example Encoding (hex):

0xB003040200001ADA

Notes:

BRANE - Branch if not equal

General usage:

BRANE hint [rC + imm32], rA, rB

Description:

The Program Counter (PC) register is replaced by the effective address found by adding the contents of register C to an immediate only if the value in register A is not equal to the value in register B.

Operation Detail:

```
if (reg (rA) == reg (rB))
   reg (PC) = reg (rC) + imm32;
```

Opcode:

0x17

Example Usage:

BRANE [r2 + 0x1ADA], r3, r4

Example Encoding (bin):

10111 ?HP 00000011 00000100 00000010 00000000_00000000_00011010_11011010

Example Encoding (hex):

0xB803040200001ADA

BRAL - Branch if less than

General usage:

BRAL hint [rC + imm32], rA, rB

Description:

The Program Counter (PC) register is replaced by the effective address found by adding the contents of register C to an immediate only if the value in register A is less than the value in register B.

Operation Detail:

```
if (reg (rA) < reg (rB))
   reg (PC) = reg (rC) + imm32;
```

Opcode:

0x18

Example Usage:

BRAL [r11 + 0x2000], r12, r13

Example Encoding (bin):

11000 ?HP 00001100 00001101 00001011 00000000_00000000_00100000_00000000

Example Encoding (hex):

0xC00C0D0B00002000

BRALE - Branch if less than or equal

General usage:

BRALE hint [rC + imm32], rA, rB

Description:

The Program Counter (PC) register is replaced by the effective address found by adding the contents of register C to an immediate only if the value in register A is less than or equal to the value in register B.

Operation Detail:

```
if (reg (rA) <= reg (rB))
   reg (PC) = reg (rC) + imm32;
```

Opcode:

0x19

Example Usage:

BRALE [r14 + 0x1BAA], r15, r1

Example Encoding (bin):

11001 ?HP 00001111 00000001 00001110 00000000_00000000_00011011_10101010

Example Encoding (hex):

0xC80F010E00001BAA

BRAG - Branch if greater than

General usage:

BRAG hint [rC + imm32], rA, rB

Description:

The Program Counter (PC) register is replaced by the effective address found by adding the contents of register C to an immediate only if the value in register A is greater than the value in register B.

Operation Detail:

```
if (reg (rA) < reg (rB))
   reg (PC) = reg (rC) + imm32;
```

Opcode:

0x1A

Example Usage:

BRAG [r11 + 0x2000], r12, r13

Example Encoding (bin):

11010 ?HP 00001100 00001101 00001011 00000000_00000000_00100000_00000000

Example Encoding (hex):

0xD00C0D0B00002000

BRAGE - Branch if greater than or equal

General usage:

BRAGE hint [rC + imm32], rA, rB

Description:

The Program Counter (PC) register is replaced by the effective address found by adding the contents of register C to an immediate only if the value in register A is greater than or equal to the value in register B.

Operation Detail:

```
if (reg (rA) <= reg (rB))
   reg (PC) = reg (rC) + imm32;
```

Opcode:

0x1B

Example Usage:

BRAGE [r14 + 0x1BAA], r15, r1

Example Encoding (bin):

11011 ?HP 00001111 00000001 00001110 00000000_00000000_00011011_10101010

Example Encoding (hex):

0xD80F010E00001BAA

INT - Generate an interrupt

General usage:

INT (rC + imm32)

Description:

Sets the Interrupt Flag Register bit for the interrupt number found by adding register C to an immediate.

Operation Detail:

```
IFR = IFR | (1 << (reg (rC) + imm32));
```

Opcode:

0x1C

Example Usage:

INT (r9 + 1)

Example Encoding (bin):

11100 ??P ???????? ???????? 00001001 00000000_00000000_00000000_00000001

Example Encoding (hex):

0xE000000900000001

IRET - Return from an interrupt

General usage:

IRET

Description:

Turns down the currently executing interrupts IFR and IRR flag, then transfers
control to next lowest priority interrupt. If no interrupts are
queued for execution then control is returned back to the main program.
IRET should be the last instruction in any interrupt routine.

Opcode:

0x1D

Example Usage:

IRET

Example Encoding (bin):

11101 ??P ???????? ???????? ???????? ????????_????????_????????_????????

Example Encoding (hex):

0xE800000000000000

5.2.8 Hinting

All instructions utilize a hint bit in the encoding; however, the hint bit only affects control instructions. The hint bit controls, when in the pipeline, the program counter gets updated with the target PC. If the hint bit is zero, then the PC does not get updated until the writeback stage. This provides for a N-1 delay slot, where N is the length of the pipeline. In the case of the MiniAT, which employs a 4 stage pipeline, then the delay slot for a false hinted branch is three. If the hint bit is set to a one, then the PC gets updated with the target address in the decode stage. Since the decode stage is the second pipeline stage in the MiniAT, then there is a one cycle delay slot.

During the implementation of a MiniAT simulator, an initial mistake in programming led to all instructions being hinted. This allowed all instructions to update the PC with the computed rC + immediate value. This let a single, non-conditional instruction write to both the PC and a separate destination register in a single instruction. One notable side effect of this is a single instruction call, which is:

```
{T} SUB r1, PC, FUNC
```

where FUNC *is* the address of the function to be called. To return from the function, implement the following:

```
{F} ADD r1, PC, (PC + sizeof(function))
```

Since the return instruction is hinted false, it has a delay slot of three instructions. These instructions could be used for any additional function call tear down procedures. The *sizeof* function returns the size of the function in words.

5.2.9 Predication

All instructions can be predicated. If the register used for predication contains a non-zero value, then during the decode stage the instruction is turned into a *NOP*.

```
add r0, r0, r0, 0
```

Figure 5.5: This is an example of a MiniAT *NOP*, but is not the only *NOP* possible

The predication bit determines which register the machine uses to determine whether or not to execute that instruction. If the register contains a 0, then the instruction is executed. If the register contains a non-zero, then the instruction is morphed into a *NOP*. Since r0 always contains a 0, (similar to SPARC, MIPS, PPC, and Itanium), then the instruction always executes.

Predication Bit Encoding	
Bit Value	**Register Used**
0	0
1	1

Predication Result	
Register Value	**Instruction Executed?**
0	Yes
1	No

5.2.10 Pseudo Instructions

MiniAT Pseudo Instructions are instructions that are called by a name that maps directly to a *single* instruction in the instruction set. For instance, a *NOP* can be implemented as an *ADD* instruction. So a *NOP is* a Pseudo instruction. During the implementation of the architecture several useful pseudo instructions were discovered and described as follows:

NEG - Negation

General usage:

NEG rA, rC

Description:

Negates the value of register C, storing the result into register register A.

Operation Detail:

SUB rA, r0, (rC + 0)

Example Usage:

NEG r4, r5

Example Encoding (bin):

00001 ??P 00000100 00000000 00000101 00000000_00000000_00000000_00000000

Example Encoding (hex):

0x0804000500000000

INVERT - Inverts a register

General usage:

INV rA, rB

Description:

Invert all bits of the value contained in register B, storing the result into register A.

Operation Detail:

```
EXOR rA, rB, (r0 + 0xFFFFFFFF)
```

Example Usage:

```
INV r7, r11
```

Example Encoding (bin):

```
01011 ??P 00000111 00001011 00000000 11111111_11111111_11111111_11111111
```

Example Encoding (hex):

```
0x58070B00FFFFFFFF
```

MOVR - Move register

General usage:

```
MOV rA, rB
```

Description:

Copies the contents of register B into register A.

Operation Detail:

```
ADD rA, rB, (r0 + 0)
```

Example Usage:

```
MOV r5, r6
```

Example Encoding (bin):

```
00000 ??P 00000101 00000110 00000000 00000000_00000000_00000000_00000000
```

Example Encoding (hex):

```
0x0005060000000000
```

MOVI - Move immediate

General usage:

```
MOV rA, imm32
```

Description:

Copies an immediate into register A.

Operation Detail:

ADD rA, r0, (r0 + imm32)

Example Usage:

MOV r3, 17

Example Encoding (bin):

00000 ??P 00000011 00000000 00000000 00000000_00000000_00000000_00010001

Example Encoding (hex):

0x0003000000000011

BRA - Unconditional branch

General usage:

BRA [rC + imm32]

Description:

The Program Counter (PC) register is replaced by the effective address found by adding the contents of register C to the immediate value.

Operation Detail:

BRAE T [rC + imm32], r0, r0

Example Usage:

BRA [r2+0x2300]

Example Encoding (bin):

10000 ?1P 00000000 00000000 00000010 00000010_00000011_00000000_00000000

Example Encoding (hex):

0x8200000202030000

NOP - No operation performed

General usage:

NOP

Description:

Exhaust a clock cycle.

Operation Detail:

ADD r0, r0, (r0 + 0)

Example Usage:

NOP

Example Encoding (bin):

00000 ??P 00000000 00000000 00000000 00000000_00000000_00000000_00000000

Example Encoding (hex):

0x0000000000000000

FLUSH - Empty the pipeline

General usage:

FLUSH

Description:

All pipeline stages are filled with *NOP*s. The PC queue is emptied, as
well, and the address of the instruction scheduled after the flush
is added back to the PC queue. This does have a delay slot of 1.

Operation Detail:

BRANE T [PC + 0], r0, r0

Example Usage:

FLUSH

Example Encoding (bin):

10000 ?0P 00000000 00000000 11111111 00000000_00000000_00000000_00000000

Example Encoding (hex):

0x800000FF00000000

5.2.11 Function Calls

The MiniAT does not have support at the instruction level for a call/return mechanism. This is one reason why delayed branches were selected; the delay slot allows for manipulation of a call stack. All calls and returns must be synthesized by the assembly programmer, assembler or compiler. There are multiple ways to achieve this mechanism, some better then others, the following sections present 3 examples slow, faster, and fastest.

Slow

This call and return mechanism simulates a traditional memory-based, push/pop, top-down stack. The "SP" points to the current top of the stack (full stack) and the stack grows down in memory.

```
; CALL !amos
       ADD     rA = PC + 6
       STORE   [SP - 1] = rA
       BRA     !amos
       SUB     SP = SP - 1
; RET
       LOAD    PC = [SP]
       ADD     SP = SP + 1
       NOP
```

Faster

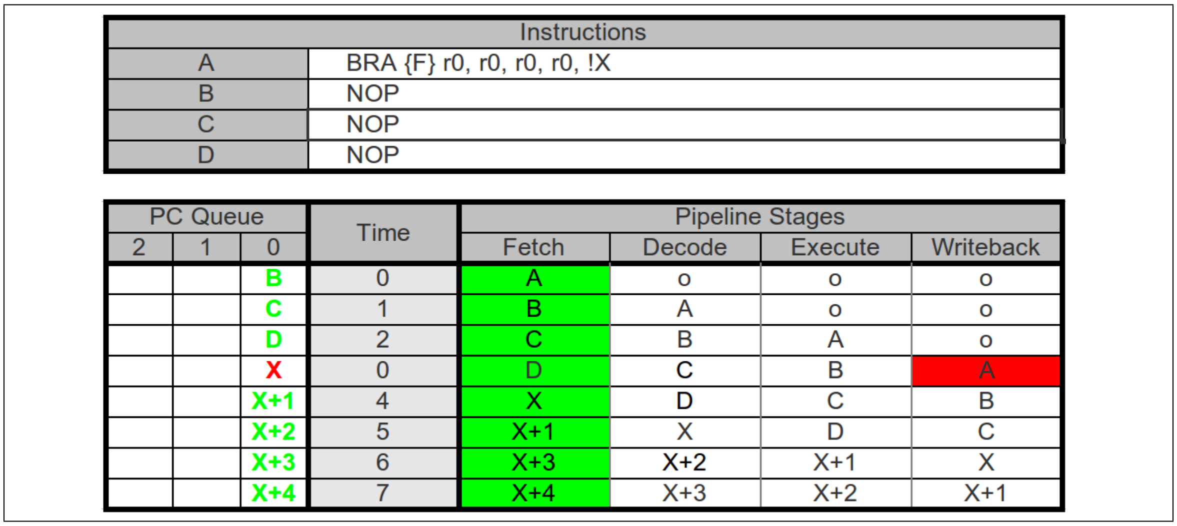
This call and return mechanism is similar to the above, but it has a reduced instruction count and uses the register stack, and thus is faster.

```
; CALL !bill
       SUB     RSP = RSP - 1
       BRA     PC = !bill
       RSTORE  [RSP] = PC
; RET
       RLOAD   PC = [RSP]
       ADD     RSP = RSP + 1
       NOP
```

Fastest

The call below is highly optimized. In fact, it used 50% fewer instructions then the second model. However, this model does not support recursion, as the “callee” cannot call any other functions, it is similar to a branch and link instruction.

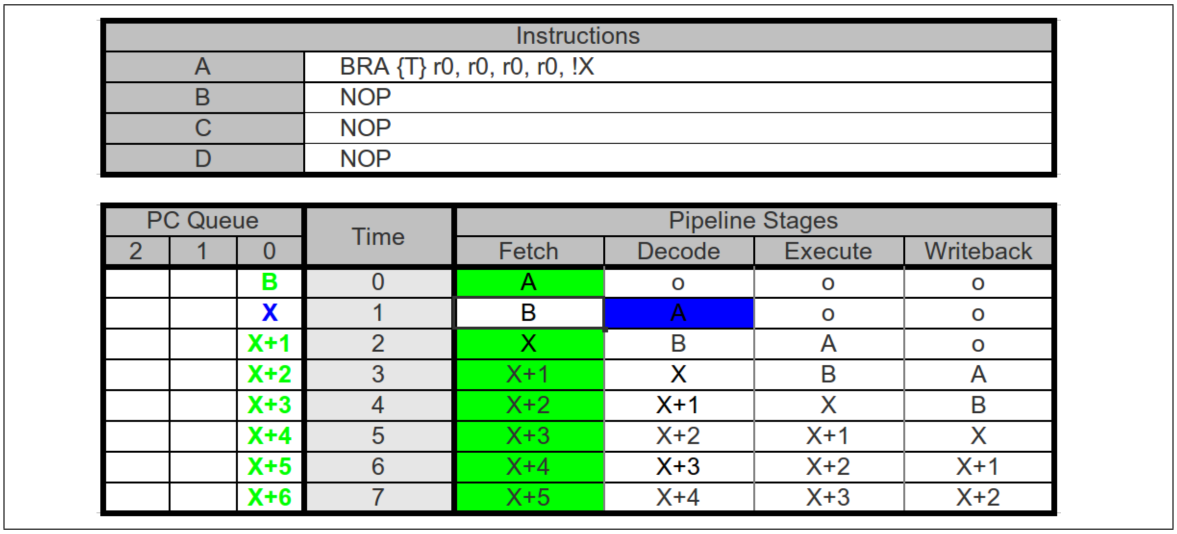
```
; CALL !rachel
       BRA     !rachel
       RSTORE  [RSP - 1] = PC
; RET
       RLOAD   PC = [RSP - 1]
```



Instructions	
A	BRA {F} r0, r0, r0, r0, !X
B	NOP
C	NOP
D	NOP

PC Queue			Time	Pipeline Stages			
2	1	0		Fetch	Decode	Execute	Writeback
		B	0	A	o	o	o
		C	1	B	A	o	o
		D	2	C	B	A	o
		X	0	D	C	B	A
		X+1	4	X	D	C	B
		X+2	5	X+1	X	D	C
		X+3	6	X+3	X+2	X+1	X
		X+4	7	X+4	X+3	X+2	X+1

Figure 5.6: Notice that the PCQ does not get updated until *time* = 3. This allows for a 3 instruction delay slot. Since there are not multiple writes to the PC in a single cycle, the queue serves no purpose here.



Instructions	
A	BRA {T} r0, r0, r0, r0, !X
B	NOP
C	NOP
D	NOP

PC Queue			Time	Pipeline Stages			
2	1	0		Fetch	Decode	Execute	Writeback
		B	0	A	o	o	o
		X	1	B	A	o	o
		X+1	2	X	B	A	o
		X+2	3	X+1	X	B	A
		X+3	4	X+2	X+1	X	B
		X+4	5	X+3	X+2	X+1	X
		X+5	6	X+4	X+3	X+2	X+1
		X+6	7	X+5	X+4	X+3	X+2

Figure 5.7: Notice that the target is written to the PCQ at time 1. This yields a 2 cycle speed improvement compared to a false hinted branch. However, in the case of a misprediction where the branch was hinted true but the boolean result was false, a branch flush occurs and execution resumes at the branches NPC.

Chapter 6

Pipeline

6.1 Overview

Pipelines provide a mechanism for the concurrent execution of instructions. Typically, an instruction requires multiple actions to happen for it to completely execute. For instance, the instruction needs to be fetched from main memory, decoded into the proper format, have its operands fetched, perform an ALU operation, and finally write the results back to some form of memory. Non-pipelined processors perform *all* the required steps for processing an instruction in either a single, long clock cycle, or multiple shorter cycles. Suppose the later scenario, for each instruction X, *in a non-pipelined processor*, it takes Y cycles for instruction X to finish. This means if it takes 4 cycles for instructions to execute, that the processor will only be finishing instruction execution at the rate of 1 instruction per 4 clock cycles. The throughput for this example is only 25%; however, a pipeline can be used to enhance throughput.

Suppose instructions need to be fetched, decoded, executed and written back. It could be modeled with a 4 stage pipeline. Each stage executes its given functionality in a clock-cycle and then passes off its output to the following stage. Initially when the machine is powered on, the pipeline will be in a state filled with some form of a *NOP* instruction. The fetch stage will start fetching instructions from wherever the program counter is initialized to. Since each pipeline stage in this example always executes within a clock-cycle, then after 4 clock cycles have passed, the pipeline will be “filled” with instructions. This is also referred to in this text as pipeline saturation. From this saturated point on, the processor produces a result *every clock cycle*. Therefore, the throughput is 100%. Generally, and under ideal situations, a pipeline of length N will provide an increase in throughput by a factor of N, relative to that machines non-pipelined version. However, this optimum is never achieved. There are a plethora of hazardous situations that arise in pipelining that reduce throughput and thus reduce performance.

Pipelines do increase performance, and this is why many modern processors feature pipelines. However, this performance increase is not "free" and requires additional architectural complexities to make it possible. The increased architectural complexity is required to handle resource dependencies because multiple stages run concurrently. Pipeline hazards are broken up into three groupings [52][34]. These groupings are:

- Structural - Issues arising from the concurrent access of mutually exclusive shared resources.
- Data - Issues arising from data dependencies between concurrent instructions.
- Control - Issues arising from manipulation of the machines program counter register.

Each one of these hazard groupings must be dealt with in a machine specific way. Anytime a hazard arises, performance can be degraded depending on the mechanism used to deal with the hazard.

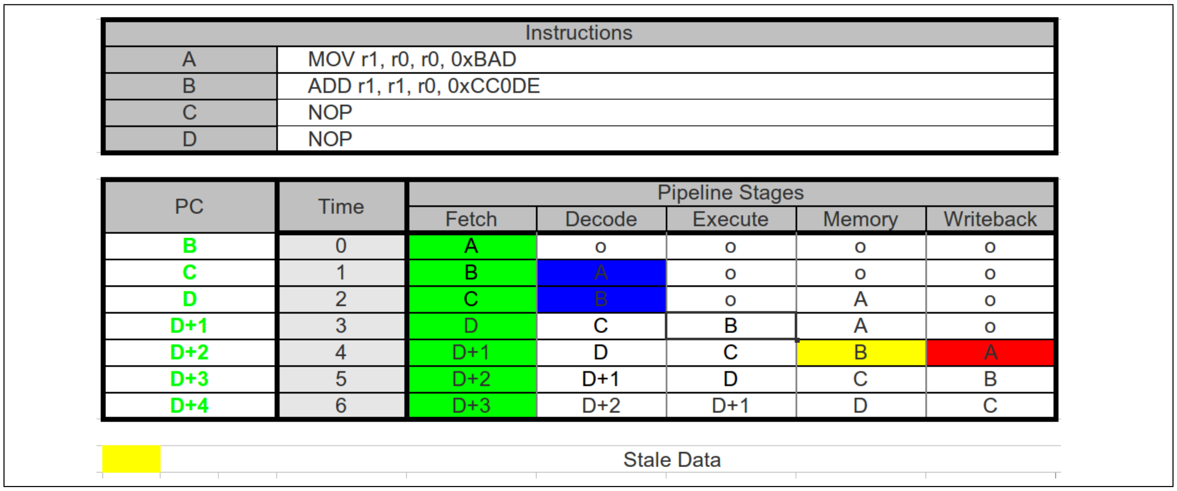
A structural hazard can arise when multiple stages are attempting to concurrently access a mutually exclusive piece of hardware. An example of this could be a Von Neumann style architecture. This architecture is one where code and data share both memory and bus. The classical RISC pipeline[34] consists of these stages:

- Stage 0 - Fetch
- Stage 1 - Decode
- Stage 2 - Execute
- Stage 3 - Memory

- Stage 4 - Writeback

Since their is only one bus and one memory set for accessing both data and instructions, a structural hazard exists when both the memory stage and fetch stage try to access memory. Only *one* stage can access memory at a time. There is an explicit memory stage, this is because most RISC architectures are of a load/store type. Therefore, only that stage accesses memory for data transactions.

Data hazards exist when a stage earlier in the pipeline requires the output of a further instruction in the pipeline that has not written the result yet (See Figure 6.1).



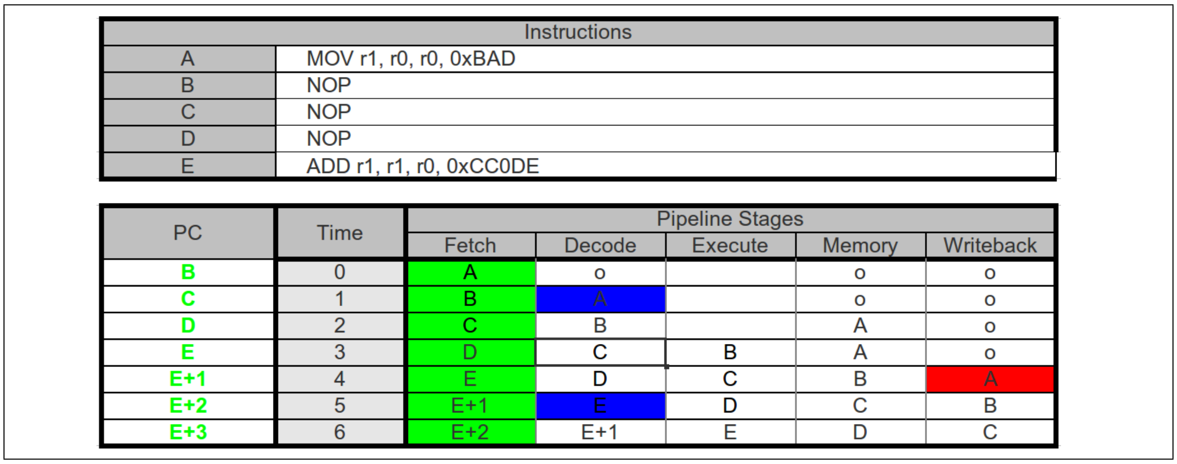
Instructions	
A	MOV r1, r0, r0, 0xBAD
B	ADD r1, r1, r0, 0xCC0DE
C	NOP
D	NOP

PC	Time	Pipeline Stages				
		Fetch	Decode	Execute	Memory	Writeback
B	0	A	o	o	o	o
C	1	B	A	o	o	o
D	2	C	B	o	A	o
D+1	3	D	C	B	A	o
D+2	4	D+1	D	C	B	A
D+3	5	D+2	D+1	D	C	B
D+4	6	D+3	D+2	D+1	D	C

🟨 Stale Data

Figure 6.1: Notice that instruction B will have stale data because A did not write its result back before B conducted an operand fetch in the decode stage.

As one can see, instruction A and B have a dependency on r1. This dependency problem can be dealt with in multiple ways[52][34][90][22]. One of the potential ways to deal with it is, not to. This approach requires the programmer or compiler to detect hazards and place enough instructions between them to break the hazards. This option allows the compiler to reorder instructions to remove data dependencies, and in the worst case add *NOP* instructions to remove hazards (See Figure 6.2). Anytime a *NOP* instruction is added, throughput decreases.



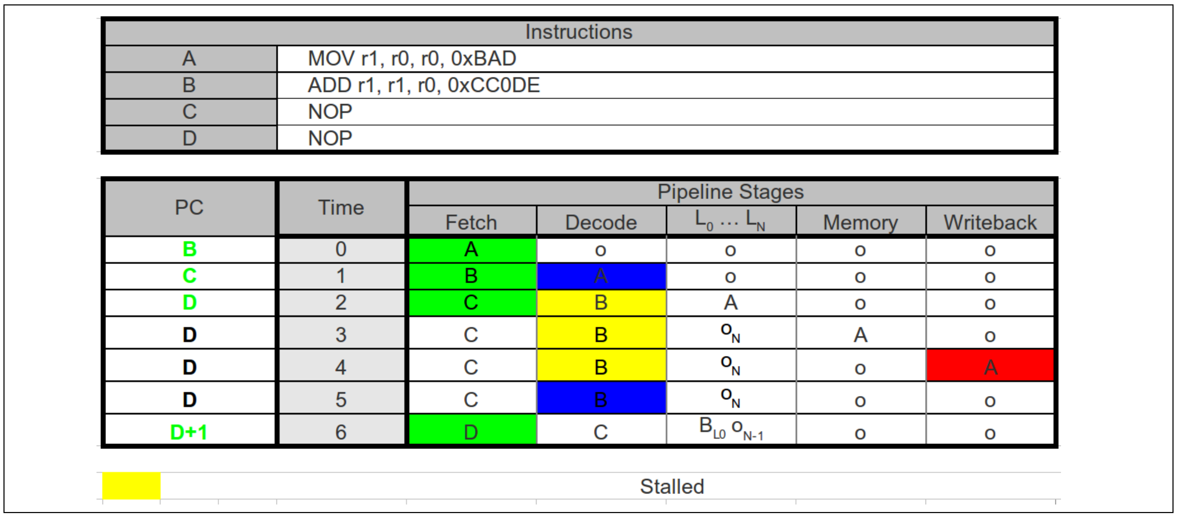
Instructions	
A	MOV r1, r0, r0, 0xBAD
B	NOP
C	NOP
D	NOP
E	ADD r1, r1, r0, 0xCC0DE

PC	Time	Pipeline Stages				
		Fetch	Decode	Execute	Memory	Writeback
B	0	A	o		o	o
C	1	B	A		o	o
D	2	C	B		A	o
E	3	D	C	B	A	o
E+1	4	E	D	C	B	A
E+2	5	E+1	E	D	C	B
E+3	6	E+2	E+1	E	D	C

Figure 6.2: Notice that *NOP*s have been inserted into the code to ensure that when instruction B performs its operand fetch, instruction A has already performed the writeback.

Another way to deal with a data hazard is called a pipeline bubble. A pipeline bubble is an instruction that cannot create data dependencies between themselves and does not alter the state of the machine. A stage earlier in the pipeline, upon detecting a data hazard, *stalls itself* and all previous stages. The stage that detected the hazard, and is currently stalling, outputs a bubble. This causes the instruction deeper in the pipeline to continue execution, and eventually eliminates the hazard by finishing execution. Once the hazard has been eliminated, the pipeline resumes normal operation. This solution provides hardware level data hazard detection and avoidance so the programmer and compiler need not be concerned. However, in this case, a data hazard always results in the execution of a bubble and thus will *always* reduce throughput. The reduction in throughput is implementation dependent.

One way to reduce the performance penalty is to have a read and write occur in the same cycle. In this scenario, the writeback writes the data early in the cycle and the operand fetch occurs late in the cycle, after the data has been written, and thus the dependency has been eliminated. This approach reduces the latency in correcting the hazard by one cycle. In the other slower alternative, the data can be written in one cycle and then in the next cycle, the operand fetch could proceed. Consider having L number of stages between the stages where the data is written and the data is fetched. In a worse case data dependency, where the instruction immediately following contains the hazard, then the number of cycles taken to clear the hazard in a same cycle read after write is L. In the slower alternative, it would be L+1.



Instructions	
A	MOV r1, r0, r0, 0xBAD
B	ADD r1, r1, r0, 0xCC0DE
C	NOP
D	NOP

PC	Time	Pipeline Stages				
		Fetch	Decode	$L_0 \dots L_N$	Memory	Writeback
B	0	A	o	o	o	o
C	1	B	A	o	o	o
D	2	C	B	A	o	o
D	3	C	B	o_N	A	o
D	4	C	B	o_N	o	A
D	5	C	B	o_N	o	o
D+1	6	D	C	B_{L0} o_{N-1}	o	o

🟨 Stalled

Figure 6.3: Notice it takes at least L cycles for instruction B to finish, relative to instruction A.

Another technique for removing data dependencies is called operand fast forwarding [52] [34] [90] [22]. Instructions are not blocked in the pipeline when data hazards are detected, but instead are allowed to continue execution. When the data is written back, that data is forwarded to the instruction that depends on those updated values. Care must be taken that under all circumstances the data can be forwarded before the machine alters state based on the old, stale data. For instance, one would want to ensure that data in the writeback was fast forwarded to a dependent instruction before any ALU operation was performed. If it did not get there prior to the ALU operation, then the result of the ALU operation would be incorrect.

Control hazards arise from manipulation of the program counter register (i.e., branching). Consider the instructions in figure 6.4. The *BRAE* instruction is executed first but the 3 instructions following are already in the pipeline when the *BRAE* instruction reaches the writeback stage. These may not be the correct instructions depending on if the branch was taken.

```
BRAE r5, r6, !BILL
ADD  r10, 0xBADCCODE
SUB  r87, r12
MUL  r1,  r8, r34, -10
```

Figure 6.4: This figure shows 4 instructions that would be in different stages of the pipeline. The instructions at !BILL may have needed to be loaded based on the branch direction.

There are two possible ways to do branching, with a delay slot, and without a delay slot. The authors will start by discussing this without delay slots. The result of the ALU comparison will not be known until after the execute stage, and the PC will not be written until the writeback stage, assuming a canonical RISC based design.

Without a delay slot, execution should transfer to target !BILL if the result of the Boolean expression is true, else one should see all instructions execute. In the case of true, all the other instructions "underneath" the branch will exist in previous stages of the pipeline, and to avoid having them execute, must be removed from the pipeline. At this point the pipeline can be reloaded with instructions from target BILL. The process of clearing the pipeline is known as a flush. The flush operation is "expensive", in a pipeline of length N, it will take at least N cycles until instructions start finishing execution again. Pipeline flushes should be avoided unless absolutely necessary. In the case of false, the branch has no effect and the other instructions would need to execute. Since they already exist in the pipeline, no action is required.

In the case of a delay slot, all instructions between the first stage of the pipeline and the stage writing to the PC are executed. This space is known as the delay slot. This avoids the expensive pipeline flush. However a performance increase is realized if and only if instructions that alter the state of the machine are placed in the delay slot. This really means that if the delay slot is padded with *NOP* instructions, it will have the same result as if the pipeline was flushed (See Figures 6.5 and 6.6).

```
ADD  r10, 0xBADCCODE
SUB  r87, r12
MUL  r1,  r8, r34, -10
BRAE r5, r6, !BILL
NOP
NOP
NOP
NOP
```

Figure 6.5: This example illustrates a delayed branch without a performance gain compared to flushing the pipeline. This example is for a 5 stage pipeline.

```
BRAE r5, r6, !BILL
ADD  r10, 0xBADCCODE
SUB  r87, r12
MUL  r1,  r8, r34, -10
NOP
```

Figure 6.6: This example illustrates a delayed branch with a 3 cycle time performance gain compared to flushing the pipeline. It also reduced the memory footprint by 63%. This example is for a 5 stage pipeline.

Branches can also be hinted [34][22], so the instruction fetch mechanism correctly predicts the correct branch to take. However, a mispredicted branch *will* result in the flush of the pipeline.

6.2 MiniAT

The MiniAT pipeline is comprised of four stages: fetch, decode, execute/memory and writeback.

6.2.1 Fetch

The fetch stage loads the instruction referenced by the PC, which is at the head of the PCQ, and removes that entry from the queue. The PC is incremented by 2 and enters the PCQ, effectively replacing the PC. The fetch stage only writes to the PCQ if the PCQ is empty.

6.2.2 Decode

The decode stage fetches register operands and calculates the immediate displacement of rC, which is used by every instruction except *IRET*. For conditional branches, the displaced rC is speculatively added to the PC queue if the Hint bit is set. If the Writeback stage is also adding to the queue, the displaced rC is added afterward.

6.2.3 Execute/Memory

The execute/memory stage uses the ALU for calculations, begins memory transfers on LOADs (and stalls until they are complete), and indirectly indexes the register file in the case of *RLOAD* and *RSTORE*.

6.2.4 Writeback

The writeback stage stores the result of the instruction. No instruction writes more than a single value. At this stage, conditional branches may flush the pipeline and replace the PCQ if the logical result of the comparison is different than the hint. If the PC is the WB target, the WB data is added to the PCQ.

By having a four stage pipeline the throughput has increased by a factor of four under ideal circumstances.

The MiniAT instruction set is divided into four logical groups, associated with those groups:

- Arithmetic and Logic
- Data (using memory)
- Data (using register file)
- Control

Figures 6.7, 6.9, 6.11, and 6.13 are given showing the flow of the pipeline. There is also discussion of example instructions moving through the pipeline given later in this section.

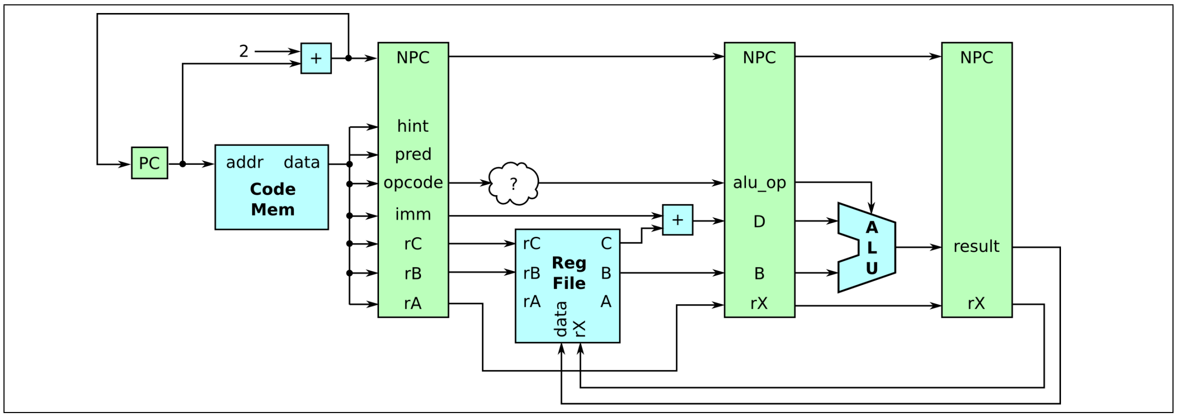


Figure 6.7: Arithmetic and logic instruction groupings can be organized into a single pipeline diagram.

For illustration, the instruction below will be described as it moves through the pipeline.

```
ADD r1, r1, r0, 1
```

Figure 6.8: This instruction adds 1 to r1 and stores the result back into r1.

- Fetch - The instruction is retrieved from memory at the location pointed to by the PC. The PC is then incremented by two.

- Decode - The instruction is broken out into the following pieces: NPC, hint, predicate, opcode, immediate, rC, rB, and rA (location of register *not* the contents). The operands are then retrieved based on the locations provided by rC and rB. Then the value contained in rC is added to the immediate and stored in the output latch as D. The register value of rB is stored to the output latch as B. NPC and rA are forwarded to the output latch and stored as NPC and rX, respectively. If the contents of the register indicated by the predication bit(r0/r1) is non-zero, then the instruction is morphed into a *NOP*.

- Execute - This stage performs the desired ALU operation; in this case *ADD*. B operator D is performed and the result is stored in the result field of the output latch. rX and NPC are also forwarded to their designated fields.

- Writeback - NPC is discarded and the result is stored in the register file at location rX.

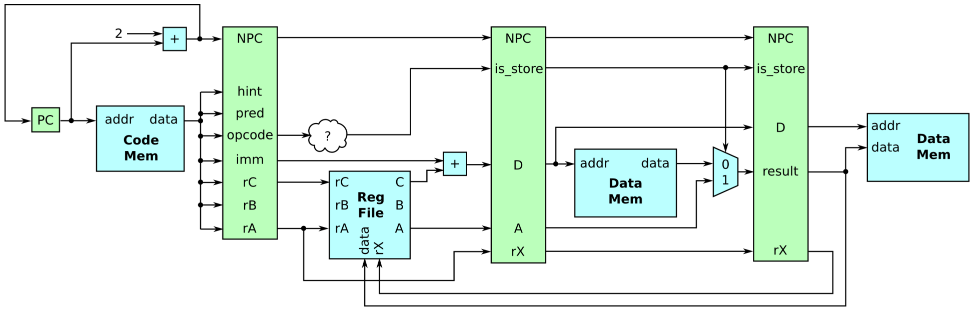


Figure 6.9: Data(using memory) instruction groupings can be organized into a single pipeline diagram.

For illustration, the instruction below will be described as it moves through the pipeline.

```
LOAD r1, r0, r0, 0x2002
```

Figure 6.10: This instruction loads r1 with the contents stored at memory location 0x2002.

- Fetch - The instruction is retrieved from memory at the location pointed to by the PC. The PC is then incremented by two.

- Decode - The instruction is broken out into the following pieces: NPC, hint, predicate, opcode, immediate, rC, and rA(location of register *not* the contents). The operands are then retrieved based on the locations provided by rC, and rA. Then the value contained in rC is added to the immediate and stored in the output latch as D. The opcode is evaluated to determine if it is a store. In the case that it is a store, a 1 is written to the is_store field of the output latch; in the other case, a 0 is stored. The register value of rA is stored to the output latch as A. rA and NPC are forwarded to the output latch field NPC and rX respectively. If the contents of the register indicated by the predication bit(r0/r1) is non-zero, then the instruction is morphed into a NOP.

- Execute - This stage performs the desired memory operation only in the case of a *LOAD*. At this point, the contents contained at memory location D are loaded into the result field of the output latch. rX, NPC, D are also forwarded to their designated fields. This stage could take more then one cycle based on memory speeds.

- Writeback - NPC is discarded, the result is stored in the register file at location rX.

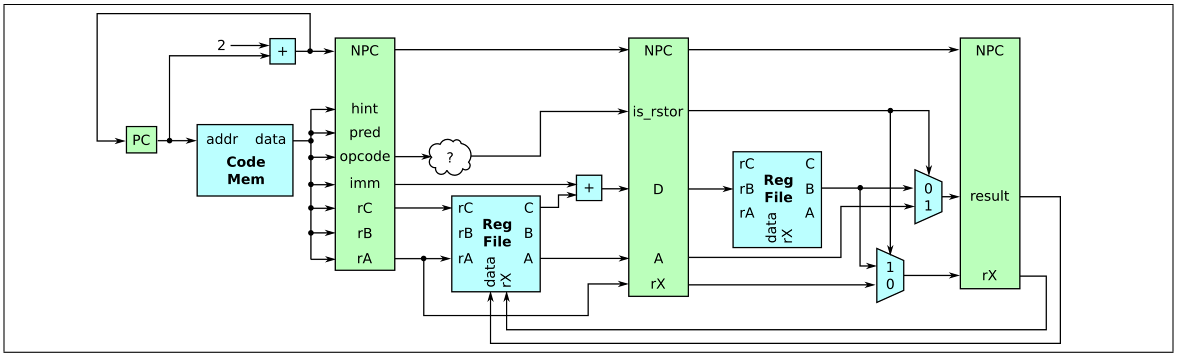


Figure 6.11: Data(using register file) instruction groupings can be organized into a single pipeline diagram.

For illustration, the instruction below will be described as it moves through the pipeline.

```
RSTOR r1, r0, r2, 0
```

Figure 6.12: This instruction stores the value in r1 to the register pointed to by r2 plus 0.

- Fetch - The instruction is retrieved from memory at the location pointed to by the PC. The PC is then incremented by two.

- Decode - The instruction is broken out into the following pieces: NPC, hint, predicate, opcode, immediate, rC, and rA (location of register *not* the contents). The operands are then retrieved based on the locations provided by rC, and rA. Then the value contained in rC is added to the immediate and stored in the output latch as D. The opcode is evaluated to determine if it is an *RSTORE*. In the case that it is an *RSTORE*, a 1 is written to the is_rstore field of the output latch. In the other case, a 0 is stored. The register value of rA is stored to the output latch as A. rA and NPC are forwarded to the output latch field NPC and rX, respectively. If the contents of the register indicated by the predication bit(r0/r1) is non-zero, then the instruction is morphed into a *NOP*.

- Execute - At this stage, D is used to access the register file and the result and A are "muxed" on the is_rstore control line. The output of the mux is stored to the result field in the output latch. B and rX are muxed on the is_rstore control line. The output of the mux is stored as the rX field in the output latch. NPC is forwarded to its designated fields.

- Writeback - NPC is discarded, the result is stored in the register file at location rX.

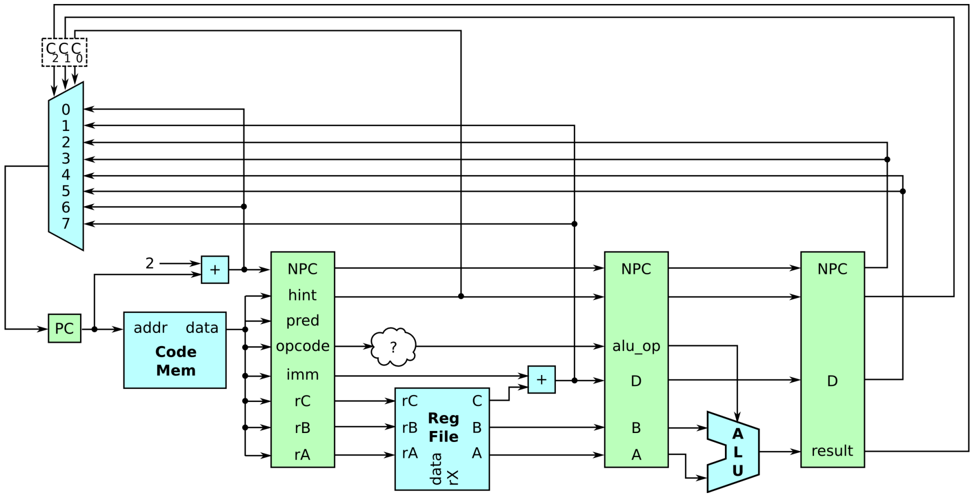


Figure 6.13: Control(conditional branches) instruction groupings can be organized into a single pipeline diagram.

For illustration, the instruction below will be described as it moves through the pipeline.

```
BRAE F r0, r0, r0, 0x2002
```

Figure 6.14: This instruction will branch to address 0x2002.

- Fetch - The instruction is retrieved from memory at the location pointed to by the PC. The PC is then incremented by two.

- Decode - The instruction is broken out into the following pieces: NPC, hint, predicate, opcode, immediate, rC, and rA(location of register *not* the contents). The operands are then retrieved based on the locations provided by rA, rB, and rC. Then the value contained in rC is added to the immediate and stored in the output latch as D. D is forwarded as a control line to a PC queue mux. The hint bit is forwarded as a control line to the PC queue mux. The register value of rA, rB, rC are stored to the output latch as A, B, and C, respectively. The opcode is forwarded as alu_op. NPC is forwarded to the output latch field NPC. If the contents of the register indicated by the predication bit(r0/r1) is non-zero, then the instruction is morphed into a *NOP*.

- Execute - At this point, the logical operation is carried out with A operator B. The output of the ALU operation is stored to the result field of the output latch. NPC, hint and D are forwarded to their respective output latch fields.

- Writeback - The result and hint bit are forwarded to the PC queue mux as control lines. D and NPC are also routed back as data lines to the PC queue mux.

There are additional complexities associated with pipelining. The MiniAT has a particular definition tied to certain verbiage that is discussed below.

6.2.5 Bubbles

When data or structural hazards are detected, bubbles can be inserted to correct the hazard. A bubble is generated by a stage outputting a *NOP* to its output latch. This allows instructions farther in the pipeline to continue and complete execution to either relieve the hazard, or provide liveliness.

6.2.6 Stalls

A stage is considered stalled when clock cycles to the stages output latch have been disabled. There are two categories of stalls:

- Active Stall - Occurs when *this* stage is stalling for a hazard. It disables the clock to the output latch, preventing the data transfer. However, a bubble is inserted into the following pipeline stage by an actively stalled pipeline stage.

- Passive Stall - Occurs when a stage deeper in the pipeline is actively stalling, preventing this stage from pushing data to its output latches.

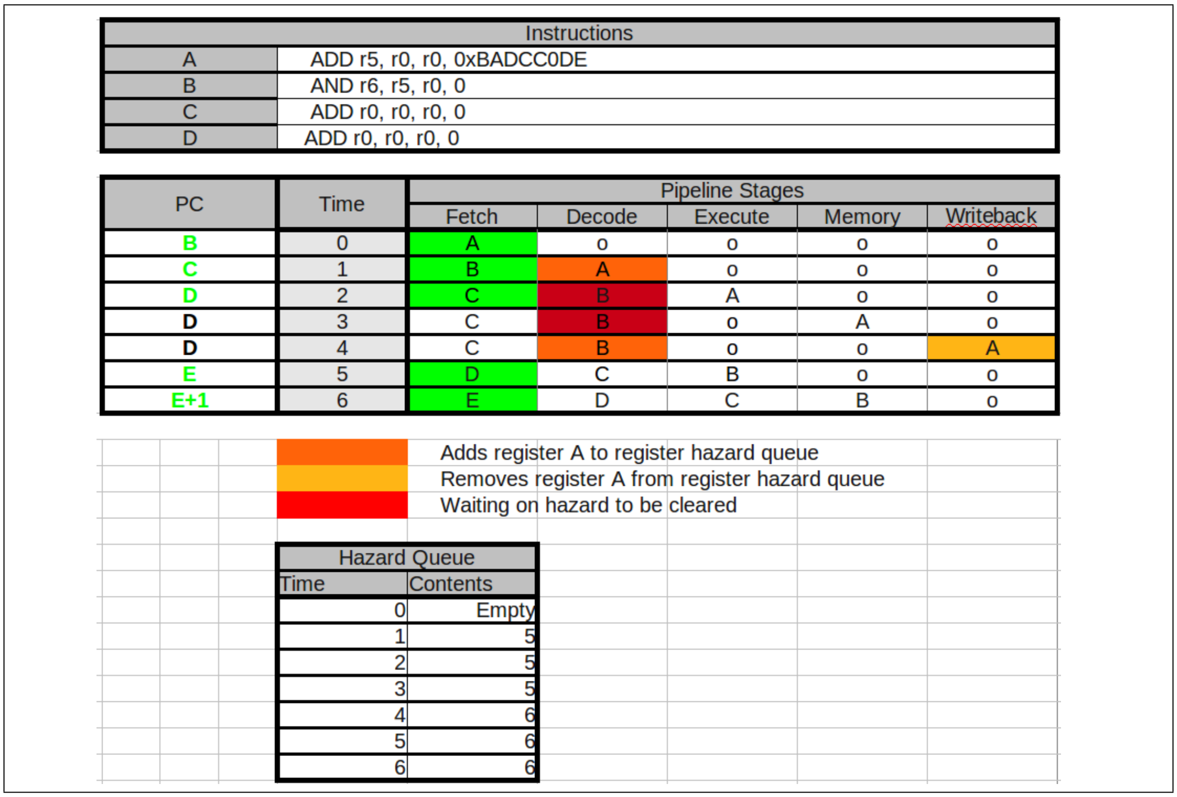
6.2.7 Hazard Buffers

The MiniAT mechanism provided at the architectural level to deal with data dependencies consists of two buffers: one for memory and the other for registers, each one having a length of 3 words. This length was derived by taking the stage where the memory or register is being marked as needing to be written, to the stage where it is actually written. In the case of the MiniAT, the decode and writeback stages mark these locations. Before an instruction can read from the register file in the decode stage, these hazard queues are checked. If the location is found in either queue, the instruction stalls until the instruction that marked the hazard finishes execution in the writeback stage.

The buffers operate on the following rules:

- In the decode stage, if a memory location or register is being written to, then its address is placed in its respective buffer. For instance, if register 5 is being written to, a 5 is placed in the register data hazard buffer. If memory location 0x2006 is being written to, then 0x2006 is placed in the memory data hazard buffer. Note: Register 0 and the PC are non-blocking and thus do not generate data dependencies. Therefore, their entries are never placed into the buffers.
- In the writeback stage, after completion of the write, the register or memory location is removed from its respective buffer.

The rest of the work is performed by the decode stage output latch. All instructions in the decode stage check their read operands against the data hazard buffers. If this address exists in the buffer, then the decode stage pushes a bubble. It continues 'pushing' bubbles, effectively pushing the data dependency to the writeback stage where it can be cleared. Once this happens, the decode stage can continue normal operation and conduct the operand fetch with the updated values (See Figure 6.15).



Instructions	
A	ADD r5, r0, r0, 0xBADCC0DE
B	AND r6, r5, r0, 0
C	ADD r0, r0, r0, 0
D	ADD r0, r0, r0, 0

PC	Time	Pipeline Stages				
		Fetch	Decode	Execute	Memory	Writeback
B	0	A	o	o	o	o
C	1	B	A	o	o	o
D	2	C	B	A	o	o
D	3	C	B	o	A	o
D	4	C	B	o	o	A
E	5	D	C	B	o	o
E+1	6	E	D	C	B	o

🟧 Adds register A to register hazard queue

🟨 Removes register A from register hazard queue

🟥 Waiting on hazard to be cleared

Hazard Queue	
Time	Contents
0	Empty
1	5
2	5
3	5
4	6
5	6
6	6

Figure 6.15: This figure shows a data dependency between two instructions and the steps required by the pipeline to prevent erroneous output.

This simple mechanism provides a powerful feature: it allows one to write code (See Figure 6.2.7) that results in the expected logical output without having to pad the binaries manually with *NOP* instructions (See Figure 6.2.7). This solution results in additional hardware complexity, but reduces the memory footprint of the executable. One should still try to avoid data dependencies, as they reduce the density of the pipeline. When pipeline density decreases so does the efficiency of the application. The example in figure 6.2.7 results in a pipeline density of 50% (See Figure 6.17). Figure 6.2.7 shows how an assembly program could deals with a data dependency. This is not necessary with the MiniAT because the architecture handles it.

```
;This instruction updates register 5 with an immediate value
;of 0xBADCC0DE

ADD r5, r0, r0, 0xBADCC0DE
;This instructions expects register 5 to already contain the
;value

AND r6, r5, r0, 0
ADD r0, r0, r0, 0
ADD r0, r0, r0, 0
```

Figure 6.16: The hardware will detect the dependency on register 5 and push bubbles to eliminate the dependency.

```
pipeline density = instructions not stalled / number of stages
```

Figure 6.17: The ideal situation would be a pipeline density of 100%. This would mean that every stage would have an instruction.

```
;This instruction updates register 5 with an immediate value
;of 0xBADCC0DE


ADD r5, r0, r0, 0xBADCC0DE
;start padding with NOPs
ADD r0, r0, r0, 0
ADD r0, r0, r0, 0
;This instructions expects register 5 to already contain the
;value


AND r6, r5, r0, 0
```

Figure 6.18: By separating the instructions that have dependencies by the number of stages that exist between the decode and the writeback stages *plus one* with other non-hazardous instructions, the machine can be alleviated from detecting these hazards. However, this results in a larger binary.

6.2.8 Structural Hazards

Structural hazards exist when multiple stages are trying to concurrently access mutually exclusive resources, or a hardware limitation requires multiple clock cycles to complete an operation. In a Von Neumann architecture, where there is a shared bus for both data and instructions, this can happen if both the fetch stage and execute/memory stage are requesting memory accesses concurrently (only one bus). By utilizing a modified Harvard architecture, the problem was avoided. However, memory accesses are still comparatively slow, and the stage making the memory request must actively stall until the request completes.

However, other structural hazards still exist. Both the execute/memory stage and the writeback stage can use the external/data bus. In this situation, the writeback stage continues execution of the bus transfer while the execute/memory stage actively stalls, thus passively stalling the previous stages in the pipeline. Once the writeback completes the request, the execute/memory stage can start its use of the bus, pushing *NOP* instructions to the writeback until it completes.

6.2.9 Control Hazards

Control hazards are when a control instruction tries to alter the flow of execution in a pipeline. This could cause a pipeline flush to ensure correct execution flow. Instructions following branch instructions are fetched before the result of the branch is known. If the branch execution path differs with the instructions already fetched, then the pipeline is flushed of all instructions and reloaded from the correct instruction address. A flush involves clearing the PC queue and setting all stages to a *NOP*.

6.2.10 Branching

The MiniAT employs hinted atomic test and branch instructions. The PC is also modifiable by all instructions that have a writable register in their format, such as *ADD*. Writes to the PC are non-blocking and do not induce data dependencies. Thus, all branches have a delay slot. The delay slot varies on the following criteria:

- Three Instruction Delay Slot - Occurs if the instruction writing to the PC is not a branch instruction, or is a false hinted branch.
- One Instruction Delay Slot - Occurs if the instruction writing to the PC is a true hinted branch.

Control instruction under certain circumstances will introduce different effects in the pipeline. The condition and effects are listed below(See Figure 6.19).

Hint	Result	Description
0	0	No flushing; execution flow does not change.
0	1	Take branch; no flushing required.
1	0	Flush the pipeline and set the PC to NPC of the instruction in writeback.
1	1	No flushing; execution flow correctly predicted.

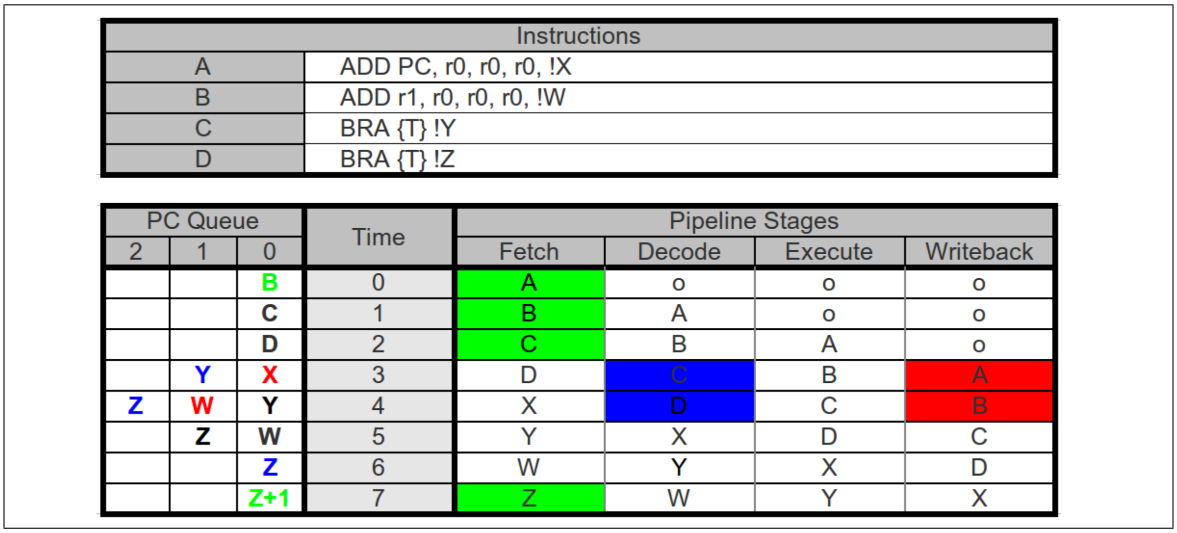
Figure 6.19: Pipeline performance could be dramatically affected based on the hint and result bit of a control instruction.

```
BRANE T r0, r0, !LABEL
NOP
```

Figure 6.20: This always results in a flush. This is also listed under pseudo instructions as the flush instruction.

6.2.11 Program Counter Queue

The MiniAT employs a program counter queue (PCQ), rather than a single program counter. Initially, it was thought this was a unique feature to the MiniAT architecture. Later in the project, it was discovered that it already exists[46]. The PCQ was utilized to address multiple writes to the program counter in a single clock cycle. Consider the following code that generates multiple writes to the PC (See Figure 6.21)



	Instructions
A	ADD PC, r0, r0, r0, !X
B	ADD r1, r0, r0, r0, !W
C	BRA {T} !Y
D	BRA {T} !Z

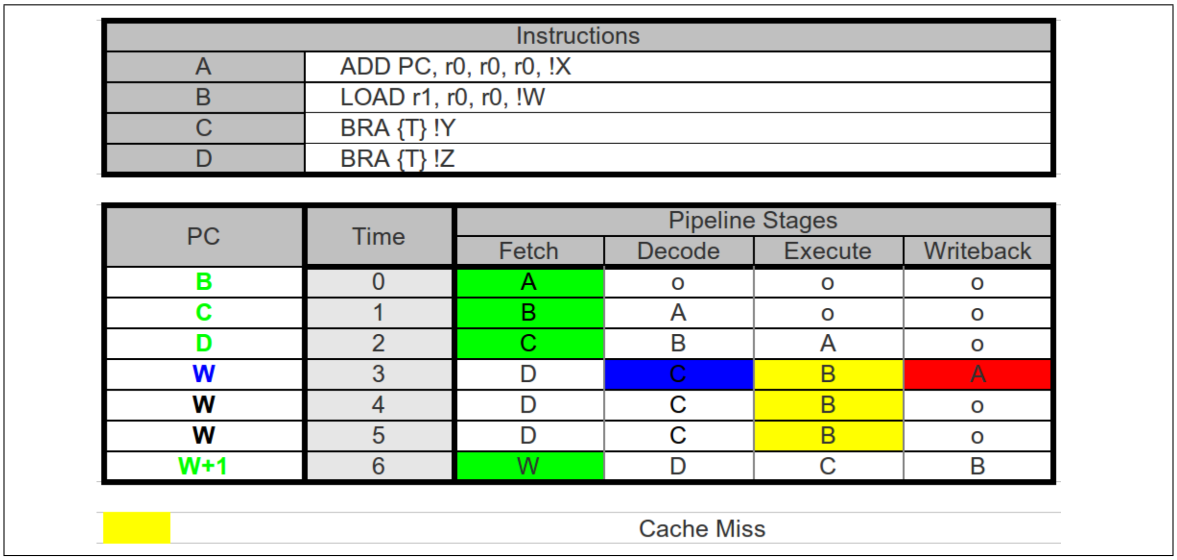
PC Queue			Time	Pipeline Stages			
2	1	0		Fetch	Decode	Execute	Writeback
		B	0	A	o	o	o
		C	1	B	A	o	o
		D	2	C	B	A	o
	Y	X	3	D	C	B	A
Z	W	Y	4	X	D	C	B
	Z	W	5	Y	X	D	C
		Z	6	W	Y	X	D
		Z+1	7	Z	W	Y	X

Figure 6.21: The example above results in the first *ADD* instruction writing to the PC in cycle 4, while at the same time the *BRAE* instruction is also writing to the PC.

Concurrent PC writes are an issue that can be solved with a PCQ. For example instruction C is hinted true, it updates the PC in the decode stage. At the same time, instruction A has made it into the writeback and is also writing the PCQ. Without some mechanism to record and correctly execute this event, one of the branches would not occur; this would be an example of a race condition. By queuing these events up, the problem can be elevated. The PCQ operates on these rules:

- Operations get queued in reverse order: the writeback adds first, the decode stage adds second, and the fetch adds last.
- The fetch stage will only add its entry if the program counter queue is empty. If the program counter queue is not empty, the stage discards its update.

A data cache miss could cause instruction B to actively stall the execute stage; thus, the decode and fetch stage would be passively stalling. This would cause A to write an X to the PC and then C to write a Y, which would overwrite A's branch target (See Figure 6.22). A possible solution to this is to stall the whole pipeline until the miss completes (See Figure 6.23). However, with the queue mechanism in place, instructions A and C place their branch targets in the queue, thus preserving their targets and their execution order, and instruction A *can* continue to writeback (See Figure 6.24). However, this mechanism does not work with instruction misses in the fetch stage. In that case, the whole pipeline must stall.

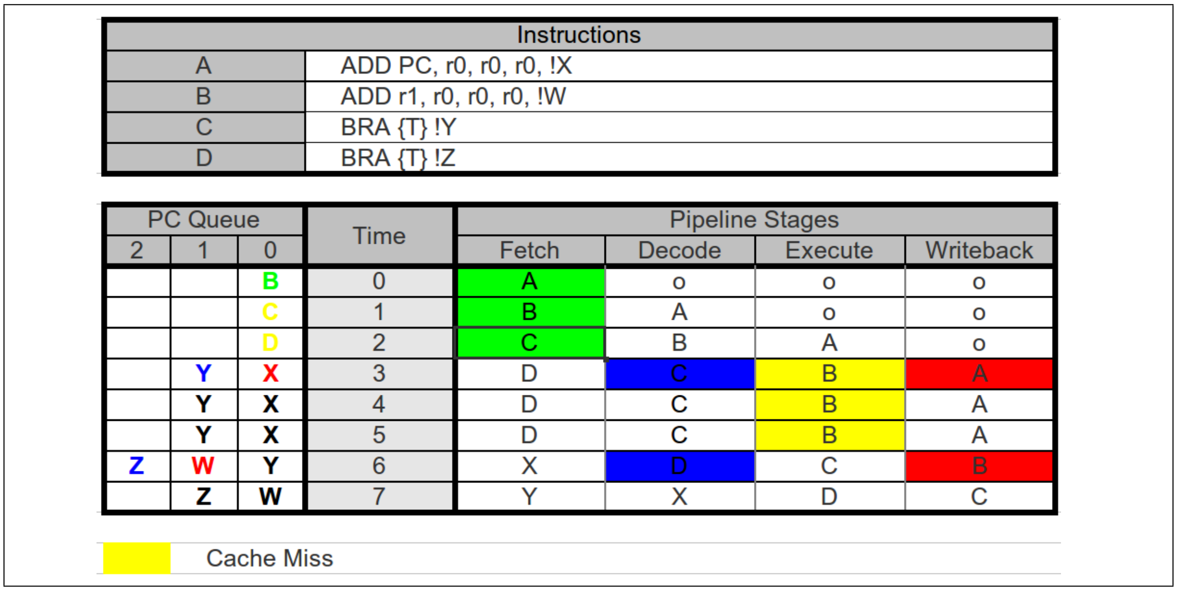


Instructions	
A	ADD PC, r0, r0, r0, !X
B	LOAD r1, r0, r0, !W
C	BRA {T} !Y
D	BRA {T} !Z

PC	Time	Pipeline Stages			
		Fetch	Decode	Execute	Writeback
B	0	A	o	o	o
C	1	B	A	o	o
D	2	C	B	A	o
W	3	D	C	B	A
W	4	D	C	B	o
W	5	D	C	B	o
W+1	6	W	D	C	B

🟨 Cache Miss

Figure 6.22: Since only one PC exists when the decode stage writes Y to the PC it overwrites the X that writeback wrote in the same cycle.

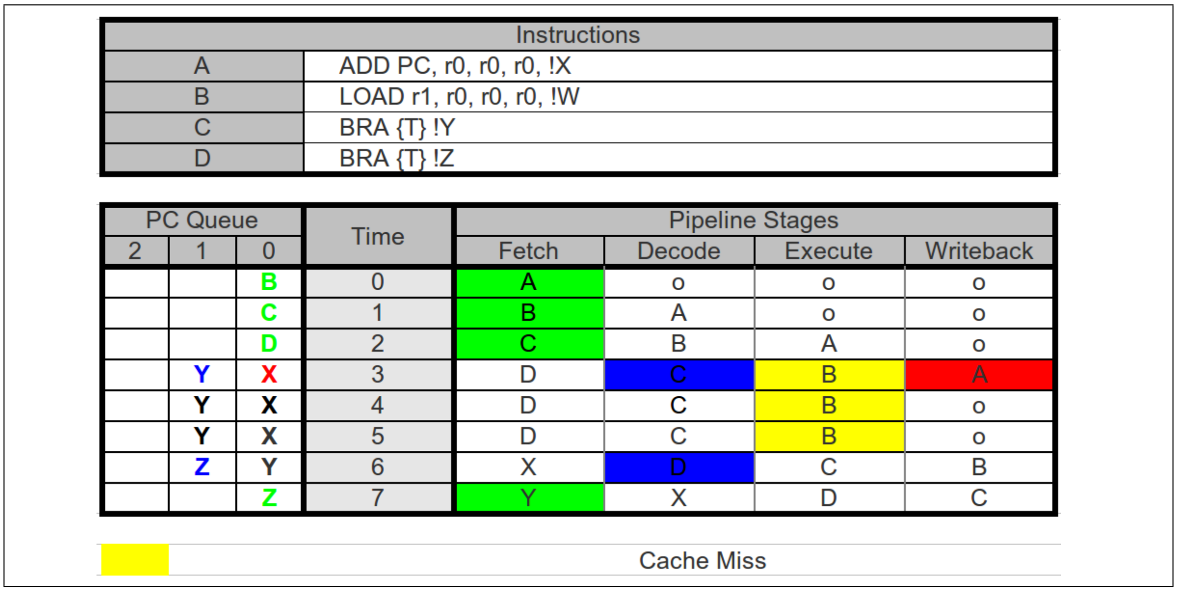


Instructions	
A	ADD PC, r0, r0, r0, !X
B	ADD r1, r0, r0, r0, !W
C	BRA {T} !Y
D	BRA {T} !Z

PC Queue			Time	Pipeline Stages			
2	1	0		Fetch	Decode	Execute	Writeback
		B	0	A	o	o	o
		C	1	B	A	o	o
		D	2	C	B	A	o
	Y	X	3	D	C	B	A
	Y	X	4	D	C	B	A
	Y	X	5	D	C	B	A
Z	W	Y	6	X	D	C	B
	Z	W	7	Y	X	D	C

🟨 Cache Miss

Figure 6.23: Since no instructions move forward, no overwriting of the PCQ occurs.



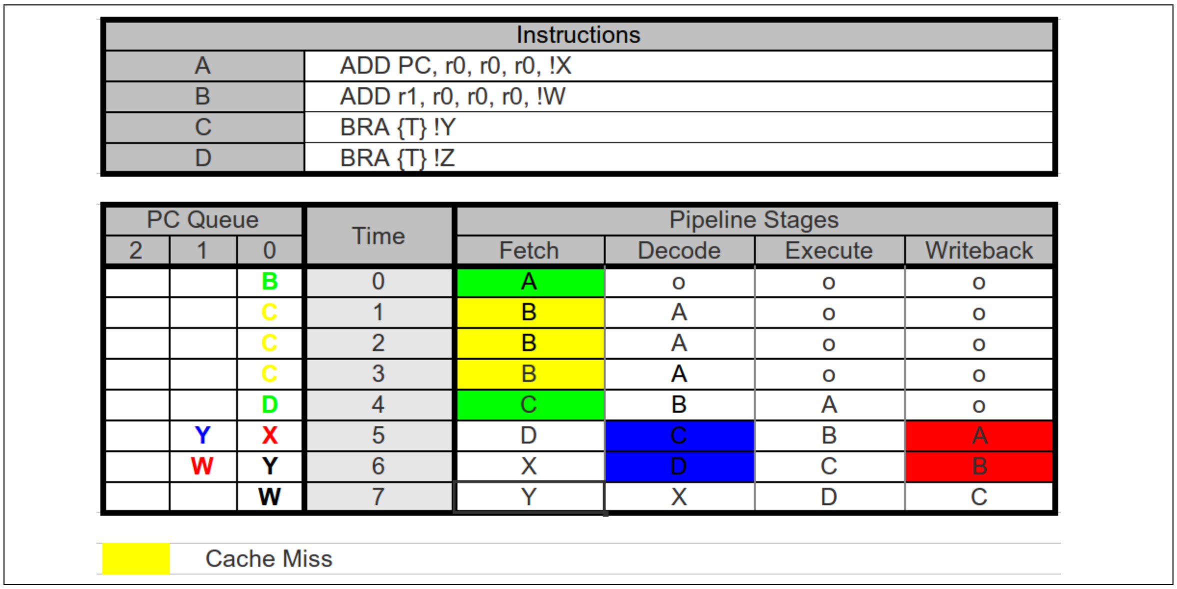
Instructions	
A	ADD PC, r0, r0, r0, !X
B	LOAD r1, r0, r0, r0, !W
C	BRA {T} !Y
D	BRA {T} !Z

PC Queue			Time	Pipeline Stages			
2	1	0		Fetch	Decode	Execute	Writeback
		B	0	A	o	o	o
		C	1	B	A	o	o
		D	2	C	B	A	o
	Y	X	3	D	C	B	A
	Y	X	4	D	C	B	o
	Y	X	5	D	C	B	o
	Z	Y	6	X	D	C	B
		Z	7	Y	X	D	C

🟨 Cache Miss

Figure 6.24: With the PCQ in place, both targets can be safely written.

Instruction Cache Miss Instruction fetch (IF) misses, caused by a data cache miss, suffer from the same issues as data cache misses. Unlike figure 6.24 where the writeback stage can continue execution, a IF miss, the pipeline must freeze. The reason for this is that as further instructions advance, the delay slot becomes filled with bubbles. If there was a branch in the pipeline, a potential delay slot would not execute correctly (See Figure 6.25 and 6.26).

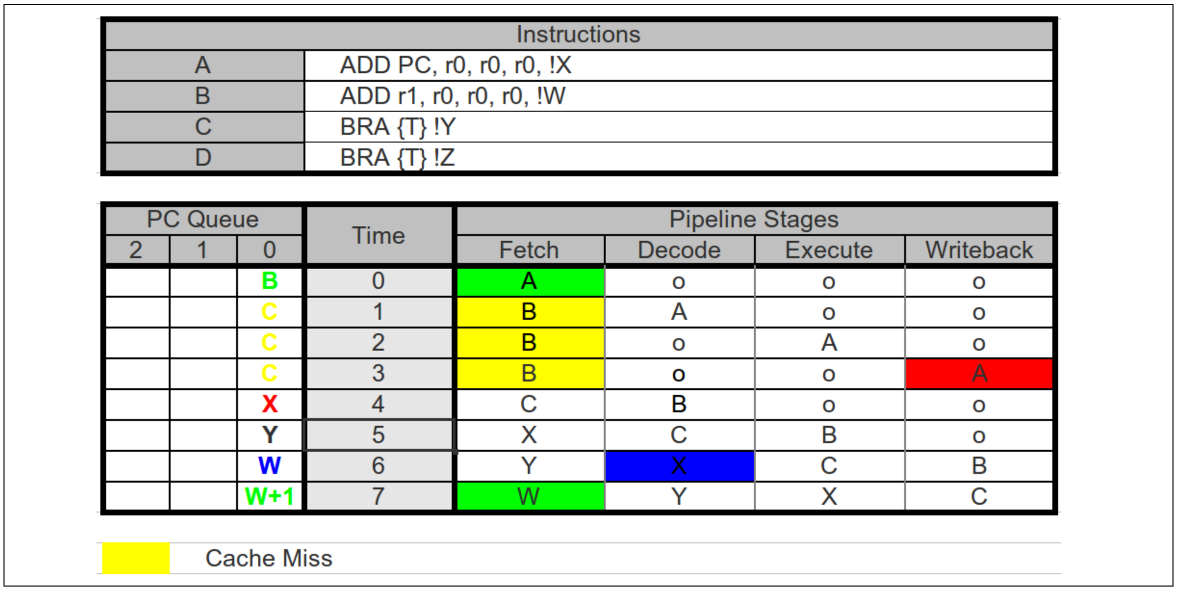


Instructions	
A	ADD PC, r0, r0, r0, !X
B	ADD r1, r0, r0, r0, !W
C	BRA {T} !Y
D	BRA {T} !Z

PC Queue			Time	Pipeline Stages			
2	1	0		Fetch	Decode	Execute	Writeback
		B	0	A	o	o	o
		C	1	B	A	o	o
		C	2	B	A	o	o
		C	3	B	A	o	o
		D	4	C	B	A	o
	Y	X	5	D	C	B	A
	W	Y	6	X	D	C	B
		W	7	Y	X	D	C

🟨 Cache Miss

Figure 6.25: Since no instructions move forward, the delay slot cannot be skewed.



Instructions	
A	ADD PC, r0, r0, r0, !X
B	ADD r1, r0, r0, r0, !W
C	BRA {T} !Y
D	BRA {T} !Z

PC Queue			Time	Pipeline Stages			
2	1	0		Fetch	Decode	Execute	Writeback
		B	0	A	o	o	o
		C	1	B	A	o	o
		C	2	B	o	A	o
		C	3	B	o	o	A
		X	4	C	B	o	o
		Y	5	X	C	B	o
		W	6	Y	X	C	B
		W+1	7	W	Y	X	C

🟨 Cache Miss

Figure 6.26: Even with the PCQ in place, the delay slot never executes.

6.2.12 Detection of Interrupts

All interrupts are handled in the writeback stage of the pipeline, this is where IFR $(AorB)| = 1 \ll X$. IFRA is used if $X < 32$, and IFRB is used if $X >= 32 \wedge x < 64$. The variable X is equal to the interrupt number thrown. Interrupts are always unsigned values; thus one can never have $X < 0$. If an instruction throws in interrupt, it is 'morphed' into an

INT instruction with the appropriate interrupt number filled out in the immediate portion of the encoding. This approach guarantees that instructions can only generate one interrupt. For example, if the divide instruction generates a divide by 0 interrupt, then the output latches for the execute stage will contain the correct contents of an *INT* 4 instruction.

The following stages can throw the interrupts listed below:

- Fetch
  - Illegal Address (INT 1) - Bad PC address.
- Decode
  - Illegal Address (INT 1) - Bad operand fetch address, checked for writes and reads.
  - Illegal Opcode (INT 2) - Malformed instruction.
- Execute
  - Divide By Zero (INT 3) - When either a divide or modulus by zero occurs.
  - Bus Error (INT 4) - When the bus state results in an error.
- Writeback
  - Bus Error (INT 4) - When the bus state results in an error.

Interrupts 5 through 10 and external interrupts 0 through 15 raise their respective IFR flag high at the end of the writeback stage.

Chapter 7

Interrupts

7.1 Overview

Interrupts provide a mechanism for the processor to halt the current execution and switch execution to some other program. This is similar to a context switch at the operating system level. Interrupts are used to drive timers, peripheral communication, system calls, arithmetic exceptions, etc. In modern processing, interrupt driven architectures are very common. Interrupts fundamentally do the same thing across processors, however their implementation is machine specific.

7.2 MiniAT

The MiniAT consists of 11 internal interrupts and 15 external interrupts. All interrupts are vector-based and must be initialized to be utilized (See Section4.2.2). Each interrupt has a corresponding enable flag located in the interrupt enable register (IER). When an interrupt is triggered, the interrupt flag register (IFR) transitions the flag of the triggered interrupt. A system wide interrupt enable flag also exists in the control status register (CSR).

The only interrupt that is not displayed in the interrupt vector table is the uninitialized interrupt. This occurs when an interrupt is enabled and fired, but contains the address of 0x0000. If this occurs, interrupt 0 (INT0) is fired by the interrupt handler, which resets the microcontroller.

7.2.1 Internal Interrupts

The microcontroller can trigger internal interrupts based on different events that take place, such as timers going off, pipeline issues and/or the *INT* instruction. The internal interrupts are:

- INT0: Reset - Restarts the microcontroller.
- INT1: Illegal Address - Fired when trying to access outside of the valid address range. An example of this would be the reserved space(0x1A00 - 0x1AD9).
- INT2: Illegal Instruction - Fired when an instruction cannot be decoded into a known operation.
- INT3: Bus Error - Fired when the bus fails transition between states correctly.
- INT4: Divide by zero - can be fired by the execute stage.
- INT5: Watchdog - Fired when the watchdog timer expires.
- INT6: Timer 0 - Fired when general purpose timer 0 completes.
- INT7: Timer 1 - Fired when general purpose timer 1 completes.
- INT8: Timer 2 - Fired when general purpose timer 2 completes.
- INT9: Timer 3 - Fired when general purpose timer 3 completes.
- INT10: Overflow - Fired when arithmetic operation computes a result that has a size greater then a word.

The following example shows how interrupts must be initialized and then causes one of them to be fired (See Figure 7.1).

```
.const IER_L 0x00001F1A
.const SYS_REG  0x00001B1E
.const IVT_ILLEGAL_ADDRESS 0x00001AE0

.address 0x00003800
!interrupt_illegal_address
ADD r1, r1, r0, 5
IRET r0, r0, r0, 0

.address 0x00002000
!main
;Initialize the Vector Table
ADD r2, r0, r0, !interrupt_illegal_address
STOR r2, r0, r0, IVT_ILLEGAL_ADDRESS

;Turn on IER_L
ADD r5, r0, r0, 0xFFFFFFFF
STOR r5, r0, r0, IER_L

;Turn on interrupts globally
ADD r6, r0, r0, 1
STOR r6, r0, r0, SYS_REG

;branch to illegal address
BRA T r0, r0, r0 ,0x0
```

Figure 7.1: The illegal address interrupt is initialized to the label *!interrupt_illegal_address* and then the IER and global interrupt flags are set to 1. After the interrupt is setup the code tries to branch to 0x0, which is an illegal address. The interrupt handler then calls the illegal address interrupt vector and passes control to !interrupt_illegal_address.

7.2.2 External Interrupts

External interrupts are just interrupts that are connected to external pins on the microcontroller. When a pin is transitioned high, the interrupt is triggered.

7.2.3 Scheduling

The interrupt handler developed by the authors and employed by the MiniAT is considered a vector based prioritized scheduling algorithm. It is very scalable in the sense that adding more interrupts to the system does not increase the complexity. The algorithm also guarantees a depth of one level in each interrupt. Also, the algorithm is implemented solely in bit-wise logic, making it very lightweight.

The scheduling algorithm has a very close relationship to OS scheduling. This is because when an interrupt occurs, the handler backs up the state of the machine and restores it on return, this is similar to the context switch performed by an OS.

The generalized interrupt handling algorithm is presented below (See Figure 7.2).

```
    1. if(SYS_REG_IE == 1 && ((IER & IFR > 0) || (IQR & IER) > 0))


        2. if(trans_bits = ((~OIFR) & IFR > 0))
            2.1. highest_int = priority_scan(tran_bits)
            2.2. IQR |= ((~IRR) & trans_bits)
            2.3. IBVT = trans_bits_enabled(all that went high)
        fi(2)


        running_int = priority_scan(IRR)
        high_queue = priority_scan(IQR)


    3. if(running_int > high_queue)
        3.1. IBVT[running_int] = PC;
        3.2. PC = IBVT[high_queue]
        3.3. IQR |= IRR
        3.4. IRR = (1 << high_queue)
        3.5. IQR ^= IRR
    fi(3)
fi(1)


OIFR = IFR
```

Figure 7.2: This generalized algorithm is able to be scaled to any size vector based interrupt system and retain its general form.

This generalized algorithm has been augmented and utilized in the MiniAT. The only additions required to the generalized interrupt handling algorithm are in Section 3 and the priority scan function (See Figure 7.3).

```
void m_interrupts_handle_interrupt(miniat *m) {

    uword *ier_l = m->interrupts.ier_l;
    uword *ier_h = m->interrupts.ier_l;

    uword *ifr_l = m->interrupts.ifr_l;
    uword *ifr_h = m->interrupts.ifr_h;

    uword *oifr_l = &(m->interrupts.oifr_l);
    uword *oifr_h = &(m->interrupts.oifr_h);

    uword *irr_l = &(m->interrupts.irr_l);
    uword *irr_h = &(m->interrupts.irr_h);

    uword *iqr_l = &(m->interrupts.iqr_l);
    uword *iqr_h = &(m->interrupts.iqr_h);

    miniat_ivbt *ivbt = m->interrupts.ivbt;

    /*these would be private to the interrupt handler*/
    uword trans_bits_l = 0;
    uword trans_bits_h = 0;
    uword running_int = 0;
    uword high_queue = 0;

    /*this checks to see if interrupts are globally enabled and
      if anything has changed state or needs to be scheduled */
    if(m->csr->csr_internals.ie && (((*ier_l & *ifr_l) > 0) ||
      ((*iqr_l & *ier_l) > 0) || ((*ier_h & *ifr_h) > 0) ||
      (*iqr_h & *ier_h) > 0)) {
```

Figure 7.3: Actual MiniAT interrupt handling algorithm implementation.

```
     trans_bits_l = *ier_l & ~(*oifr_l) & *ifr_l;
     trans_bits_h = *ier_h & ~(*oifr_h) & *ifr_h;
/*if anything transitioned it needs some scheduling*/
if(trans_bits_l || trans_bits_h) {

     /*Load the ivbt with information from the ivt*/
     m_interrupts_load_ivbt(m, trans_bits_l, trans_bits_h);
     /*queue everything that isn't already running and that
       transitioned over*/
     *iqr_l |= ~(*irr_l) & trans_bits_l;
     *iqr_h |= ~(*irr_h) & trans_bits_h;
}
/*Determine which is the highest interrupt in the running
  and queued registers*/
running_int = m_interrupts_priority_scan(*irr_l, *irr_h);
high_queue = m_interrupts_priority_scan(*iqr_l, *iqr_h);

/*if the running interrupt has a lower priority than a
  queued interrupt, schedule the queued interrupt*/
if(running_int > high_queue) {
     /*
      * Back up the state of the current pipeline
      */
     *(ivbt[running_int].pc.queue[0]) =
                            *(m->pipeline.pc.queue[0]);

     *(ivbt[running_int].pc.queue[1]) =
                            *(m->pipeline.pc.queue[1]);
```

Figure 7.4: Actual MiniAT interrupt handling algorithm implementation continued.

```
*(ivbt[running_int].pc.queue[2]) =
                       *(m->pipeline.pc.queue[2]);

ivbt[running_int].pc.size =  m->pipeline.pc.size;

 memcpy(&(ivbt[running_int].hazards),
 &(m->pipeline.hazards), sizeof(m->pipeline.hazards));

memcpy(&(ivbt[running_int].decode),
         &(m->pipeline.decode.in),
                    sizeof(m->pipeline.decode.in));

memcpy(&(ivbt[running_int].execute),
         &(m->pipeline.execute.in),
                    sizeof(m->pipeline.execute.in));

memcpy(&(ivbt[running_int].writeback),
         &(m->pipeline.writeback.in),
                    sizeof(m->pipeline.writeback.in));

/*
 * Clear out the pipeline by flushing it
 */
m_pipeline_flush(m);

/*
 * Set the pipeline equal to what is stored at
 * IVBT[high_queue]
 */
*(m->pipeline.pc.queue[0]) =
                 *(ivbt[high_queue].pc.queue[0]);
```

Figure 7.5: Actual MiniAT interrupt handling algorithm implementation continued.

```
               *(m->pipeline.pc.queue[1]) =
                                *(ivbt[high_queue].pc.queue[1]);

               *(m->pipeline.pc.queue[2]) =
                                *(ivbt[high_queue].pc.queue[2]);

               m->pipeline.pc.size = ivbt[high_queue].pc.size;

               memcpy(&(m->pipeline.hazards),
                        &(ivbt[high_queue].hazards),
                                   sizeof(m->pipeline.hazards));

               memcpy(&(m->pipeline.decode.in),
                        &(ivbt[high_queue].decode),
                                   sizeof(m->pipeline.decode.in));

               memcpy(&(m->pipeline.execute.in),
                        &(ivbt[high_queue].execute),
                                   sizeof(m->pipeline.execute.in));

               memcpy(&(m->pipeline.writeback.in),
                        &(ivbt[high_queue].writeback),
                                   sizeof(m->pipeline.writeback.in));

               *iqr_l |= *irr_l;
               *iqr_h |= *irr_h;
               /*need to split up the bit shifting by word*/
}
```

Figure 7.6: Actual MiniAT interrupt handling algorithm implementation continued.

```
if(high_queue < (sizeof(uword) * 4)) {
                    *irr_l = (1 << high_queue);
               }
               else {
                    *irr_h = (1 << (high_queue -
                                          (sizeof(uword) * 4)));
               }
               /*turn off who is running from the queue*/
               *iqr_l ^= *irr_l;
               *iqr_h ^= *irr_h;
          }
     }
     if(*ifr_l & 1 << M_INT_RESET) {
          miniat_reset(m);
}
return;

/* this function should take in a two 32 bit registers representing 64
 * bit word and return the index of the lowest bit turned on
 */
int m_interrupts_priority_scan(uword in_register_l,
                                             uword in_register_h) {
int n = 1;
if(in_register_l== 0 && in_register_h == 0) { return 64; }

        if(in_register_l == 0) {
             n = n+32; in_register_l = in_register_h;
        }
```

Figure 7.7: Actual MiniAT interrupt handling algorithm implementation continued.

```
if((in_register_l & 0x0000FFFF) == 0) {
             n=n +16; in_register_l = in_register_l >>16;
        }
if((in_register_l & 0x000000FF) == 0) {
             n=n +8; in_register_l = in_register_l >>8;
        }
if((in_register_l & 0x0000000F) == 0) {
             n=n +4; in_register_l = in_register_l >>4;
        }
if((in_register_l & 0x00000003) == 0) {
             n=n +2; in_register_l = in_register_l >>2;
        }
return n - (in_register_l & 1);
}


/* this function will load the ivbt with the ivt for all interrupts
 * that transitioned high
 */
void m_interrupts_load_ivbt(miniat *m, uword trans_bits_l,
                                                uword trans_bits_h) {


     int i=0;
     int offset=0;
     for(i=0; i<(sizeof(uword) * 4); i++) {
          if((trans_bits_l >> i) & 1) {
               *(m->interrupts.ivbt[i].pc.queue[0]) =
                                       m->mem[i + M_IVT_START];
               *(m->interrupts.ivbt[i].pc.queue[1]) = 0;
               *(m->interrupts.ivbt[i].pc.queue[2]) = 0;
               m->interrupts.ivbt[i].pc.size = 1;
```

Figure 7.8: Actual MiniAT interrupt handling algorithm implementation continued.

```
               memset(&(m->interrupts.ivbt[i].decode), 0x0,
                           sizeof(m->interrupts.ivbt[i].decode));

               memset(&(m->interrupts.ivbt[i].execute), 0x0,
                           sizeof(m->interrupts.ivbt[i].execute));

               memset(&(m->interrupts.ivbt[i].writeback), 0x0,
                           sizeof(m->interrupts.ivbt[i].writeback));
          }
     }
     for(i=0; i<(sizeof(uword) * 4); i++) {
          if((trans_bits_h >> i) & 1) {
               offset = i+(sizeof(uword) * 4);
               *(m->interrupts.ivbt[offset].pc.queue[0]) =
                               m->mem[offset+M_IVT_START];
               *(m->interrupts.ivbt[offset].pc.queue[1]) = 0;
               *(m->interrupts.ivbt[offset].pc.queue[2]) = 0;
               m->interrupts.ivbt[offset].pc.size = 1;

               memset(&(m->interrupts.ivbt[offset].decode), 0x0,
                           sizeof(m->interrupts.ivbt[offset].decode));

               memset(&(m->interrupts.ivbt[offset].execute), 0x0,
                           sizeof(m->interrupts.ivbt[offset].execute));

               memset(&(m->interrupts.ivbt[offset].writeback), 0x0,
                           sizeof(m->interrupts.ivbt[offset].writeback));
          }
     }
     return;
}
```

Figure 7.9: Actual MiniAT interrupt handling algorithm implementation continued.

Chapter 8

System Control Registers

8.1 Overview

Processors support multiple options and features. These features and options are controlled through flag settings in registers. The options controlled are *always* architecture dependent.

8.2 MiniAT

The MiniAT uses two registers for controlling processor options. There is a single global register for enabling/disabling all features, and since interrupts are both globally and individually maskable, a separate register for manipulating individual interrupts is required.

8.2.1 System Register

The System Register (at address 0x1B1E) contains several CPU level control bits as follows and illustrated in Figure 8.2.1:

- Interrupt Enable (IE) globally enables or disables interrupts (except Reset) without regard or change to any Interrupt Enable Register.
- Saturation Enable (SE) prevents overflow. When set, signed and unsigned results are clipped to 0x8000 and 0x0000 respectively and saturated at 0x7FFF and 0xFFFF respectively.

- Unsigned (U) forces the ALU and the register displacement adder in the decode stage of the pipeline to calculate results based on unsigned operands/operations and affects saturation if SE is set.

- When set, Flush on IER write (FI) forces a pipeline flush whenever the writeback stage of the pipeline manipulates either the IERA or IERB registers.

- When set, Flush on System register write (FS) forces a pipeline flush whenever the writeback stage of the pipeline manipulates the system register.

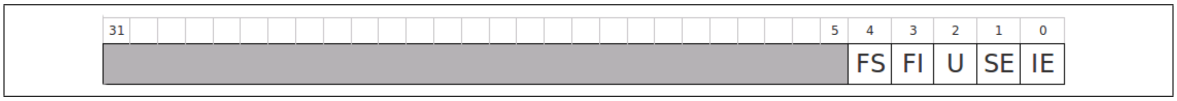


Figure 8.1: MiniAT System Register Layout

Manipulations of the System Register immediately alter the functionality of various stages in the pipeline and might, therefore, inappropriately affect instructions already fetched. To avoid these problems, one should take care where in the instruction stream the System Register is modified or flush the pipeline and PCQ immediately following such modifications with a *FLUSH* instruction. This instruction purposefully mispredicts branching to the next instruction. Thus, when it reaches the writeback stage of the pipeline it flushes all previously fetched instructions/data and replaces the PCQ with the original next line of code. This effectively delays the runnable fetch of the instruction following the mispredicted branch by four cycles.

8.2.2 Interrupt Enable Registers

The Interrupt Enable Registers (IER) are contained across two consecutive memory addresses: IERA is at address 0x1F1A and IERB is at address 0x1F1B. IERA is used for manipulating maskable internal interrupts; however interrupt 0 (reset) is not maskable. By placing a 1 in the bit position that is equal to the interrupt number, that interrupt will be enabled. This assumes that the global interrupt enable bit is high in the system register.

The same holds true for IFRB, except it is for external interrupts. Since external interrupts start at number 32, one must subtract 32 from the interrupt number when trying to disable or enable and add 1 to arrive at the correct bit location. For instance, if one wishes to enable interrupt 36, the assembly code in figure 8.2 would be written.

```
.const IERB 0x1F1B

.const MASK 010000b

;Notice bit position 5 is being flagged high, 32 + 5 - 1 = 36

;Which can be arranged as: 36 - 32 + 1 = 5

LOAD r2, r0, r0, IERB

OR   r2, r0, r0, MASK

STOR r2, r0, r0, IERB
```

Figure 8.2: This code shows how to enable interrupt 36, by creating a mask and using the or operator to update the copy of IERB loaded into r2. The result is then stored back to the IERB memory location to cause the interrupt to become enabled.

8.2.3 Interrupt Flag Registers

The Interrupt Flag Registers (IFR) are contained across two memory locations. IFRA is at address 0x1F1C and IFRB is at address 0x1F1D. IFRA is used to control internal interrupts and IFRB is used for controlling external interrupts. By setting a bit position X high, interrupt X is tripped. This is how the *INT* instruction operates. However, since they span multiple registers, interrupts greater than 31 are in IFRB. To enable interrupts in IFRB one must follow a similar approach to enabling an interrupt in IERB (See Figure 8.2).

Chapter 9

Bus

9.1 Overview

Bus structures are ways for devices to communicate. Many bus structures exist; some are typically reserved for internal memory organization and others for external connections such as:

- Von Neumann
- Harvard
- Modified Harvard
- I2C
- SPI
- PBI

The Von Neumann architecture organizes memory with shared code and data. It uses a single bus to access both types of stored data.

The Harvard architecture uses two physically separate memory chips: one for data and another for code. This architecture uses a separate bus for each memory module.

The Modified Harvard architecture leverages features of both the Harvard and Von Neumann architectures. It utilizes a single memory chip with two separate buses: one for code and another for data access.

Other buses are typically reserved for external connections to peripherals. Examples of these buses include:

The I2C [84] is a 2 wire, synchronous bus protocol. The bus has a master/slave relationship and all nodes on the bus must be addressed in a multi-slave multi-master setup. In a single master multi or single slave setup, only the slaves need addresses. There are error correction and detection schemes as well as bus arbitration schemes implemented in the hardware.

SPI[91] is a 4 or 3 wire serial, low power, communication bus. It allows for full duplex communication. Unlike the I2C bus, there is no hardware level error detection.

The PBI bus[16] is a 50 pin parallel communication protocol found on Atari systems. Notable pins that influenced the MiniAT design were the address, data, and read/write.

9.2 MiniAT

9.2.1 Internal Bus

Accesses to on-chip memory devices are through a modified Harvard architecture. The code bus is divided into 2 segments: 16 bits for the address and 64 bits for the instruction. The data bus is also divided into 2 segments: 16 bits for the address and 32 bits for the data. Both the instruction and data buses employ separate caches.

9.2.2 External Bus

The MiniAT bus is a 51 pin, asynchronous, half-duplex, return to zero parallel bus system. It specifies no data error detection or correction schemes, and it does not specify a bus arbitration policy. It is a single master, multi-slave protocol where all slaves must have designated addresses on the bus. Since bus addresses are from 0x4000 and to 0xFFFF, there is support for 49152 distinct addressable nodes. The bus is broken down into 3 main parts, which are:

- Data - 32 bit wide, with bit 0 being the least significant bit of the least significant byte in the word.
- Address - The address is 16 bits wide, with bit 0 being the least significant bit of the least significant byte in the wyde.
- Control - Their are 3 control lines. The request line (REQ), the acknowledgment line (ACK) and the read or write line (r/W). When the r/W pin is low it indicates a read.

To illustrate how a bus transfer would act, two figures are presented below: one that describes a read operation (See Figure 9.1) and another that shows a write operation (See Figure 9.2). The different state transitions as devices in control at a particular state are identified.

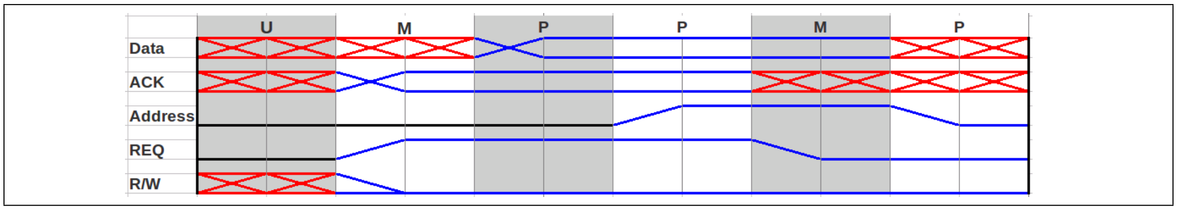


Figure 9.1: Master reading from the MiniAT bus. Only the MiniAT can initiate a request. The 'Ms' and 'Ps' at the top correspond to the MiniAT and Peripheral states, respectively, with time progressing to the right. The pictures are modifications of originals developed by Dr. William Confer.

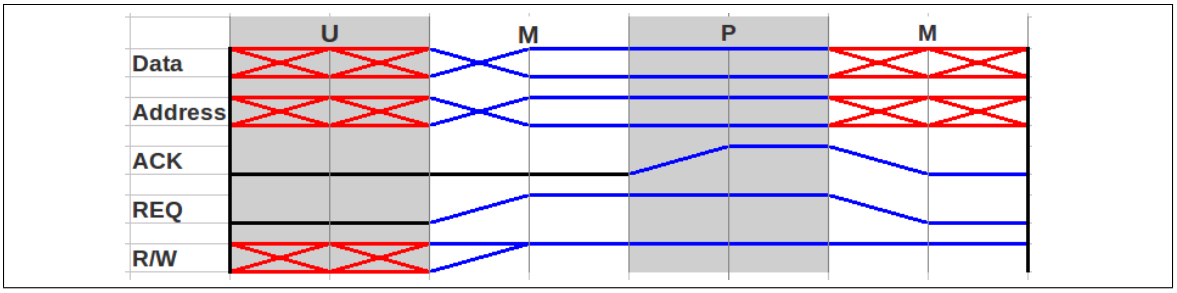


Figure 9.2: Master writing to the MiniAT bus. Only the MiniAT can initiate a request. The 'Ms' and 'Ps' at the top correspond to the MiniAT and Peripheral states, respectively, with time progressing to the right. The pictures are modifications of originals developed by Dr. William Confer.

Since there is no guaranteed length of time for peripheral transactions to occur in, bus transactions can hang the device. It is also a good idea to disable all interrupts while doing bus transactions. If an interrupt is introduced during a bus transaction, the bus would be in some arbitrary state Y and, upon returning from the interrupt, it could be in a different state. If this happens, the instruction attempting the bus access needs to starts over, and expects the bus in a low state across all pins (return to 0). This state cannot be guaranteed through the bus protocol itself.

Since the bus has no timeouts, a bad peripheral transaction could hang the entire machine. It would then be good practice to enable the watchdog interrupt while doing bus transactions. It is also good practice to leave the bus error interrupt enabled. However, this leads back to the problem of interrupts occurring during bus transactions. An external solution does exist. By providing a GPIO line to all peripherals on the bus, this "broadcast" line could act as a signal that an interrupt has occurred. All slaves upon receiving this interrupt would be required to re-enter the initial state, the state that all load and stores expect. A side effect of this is that all interrupt vectors would be required to send this broadcast message out to peripherals upon entering the interrupt vector. Due to possible configuration differences, this solution would not be considered portable as different systems could use different GPIO ports or even different mechanisms.

Another issue with using the bus is that the bus does not provide a way for interrupt driven I/O directly. Instead one can use a GPIO line can be attached to a peripheral external interrupt pin to wake the peripheral to write to it. The opposite can be accomplished for the MiniAT. By attaching a line from an output pin on the peripheral to an external interrupt pin on the MiniAT, the peripheral can wake the master for a specified task.

Chapter 10

Timers

10.1 Overview

Timers provide a means for computers to halt execution and perform a specific task at a desired interval. For instance, the watchdog timer is used in conjunction with an operating system to provide tasks with a time slice. When tasks are placed onto the CPU by the OS, they are only allowed to run for a specific time. This is knows as a time slice or quantum. This is just one usage of a timer. Timers can also be used to output frequencies, perform analog to digital conversions, PWM control stepper motors, etc.

10.2 MiniAT

The MiniAT has two types of timers: the watchdog timer and general timers. Each timer, including the watchdog, has an associated interrupt it triggers (See Section 4.2.2).

10.2.1 Watchdog Timer

The watchdog timer consists of two registers and an interrupt flag that will be triggered if the watchdog fires. Figures 10.1, and 10.2 present the layout of both registers.

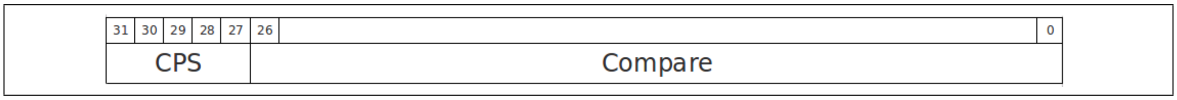


Figure 10.1: MiniAT Watchdog Compare

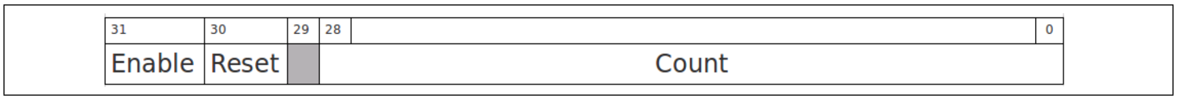


Figure 10.2: MiniAT Watchdog Count

A few steps are required for setting up the watchdog timer, which are as follows:

- The watchdog timer will set the IFRA bit 6 to a 1 (or cause Reset INT0 if the watchdog vector is NULL) when enabled (Watchdog Count→Enable is set to 1) and Watchdog Count→Count ¿= Watchdog Compare→Compare. Software must keep the Watchdog Count→Count low to prevent the interrupt from firing.

- Watchdog Count→Count is incremented once every 2^{CPS} clock ticks, where CPS (Clock Pre-Scale) is the 5 bit Watchdog Compare→CPS, managed by a private modulus counter. This provides watchdog periods up to 2.8∗1017 clock ticks... approximately 91 years for a 100MHz clock rate.

- Watchdog Count→Enable (dis)allows clock ticks to affect the modulus counter, but will not reset this counter when reset to 0

- Watchdog Count→Reset will disable the watchdog timer (Watchdog Count→Enable set to 0), set the modulus counter to 0, and Watchdog Count→Count to 0.

10.2.2 General Purpose Timers

4 general purpose timers are available in the architecture. Each behaves in one of the following four modes:

- Time

- Count

- Toggle

- PWM

The timers utilizes 3 registers: Compare, Control, and IO. Each timer also controls a single interrupt.

The compare register is shared by all timers and has a byte dedicated to each that contains the CPS (clock pre-scale), reset, mode, and enable (See Figure 10.3).

31	16	15	0
Frequency		Duty	

Figure 10.3: Timer Compare X Layout

Every timer has a dedicated control register which contains the frequency and duty (See Figures 10.4, 10.5, 10.6, 10.7).

Each mode (except count) uses a pre-scaled clock to signal each operation: the $pre-scaledclockrate = Clockrate/(CPS+1)$, where CPS is a 4 bit value defined separately for each timer as Timer Control→CPS X. This pre-scaled rate is achieved by modulus counting of clock ticks via private registers, one for each timer. When the register is cleared (reset to 0) by the modulus counter, a pre-scaled clock tick occurs.

Timer operations are "pausable". Disabling a timer only prevents clock ticks from affecting the modulus counter so that operation might be enabled ("resumed") from the original disabled state. Only a timer reset operation ($TimerControl \rightarrow ResetX = 1$) will guarantee the modulus counter is initialized to 0. Timers are pre-loadable (regardless of mode) by writing to Timer X Count.

7	6	5	4	3	0
Reset	Mode		Enable	CPS	

Figure 10.4: MiniAT Timer Control 0

15	14	13	12	11	8
Reset	Mode		Enable	CPS	

Figure 10.5: MiniAT Timer Control 1

23	22	21	20	19	16
Reset	Mode		Enable	CPS	

Figure 10.6: MiniAT Timer Control 2

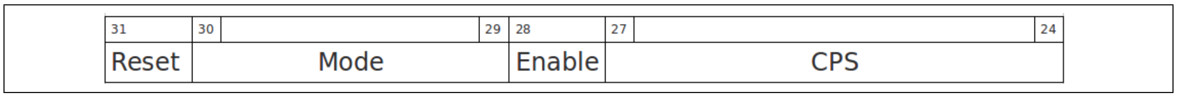


Figure 10.7: MiniAT Timer Control 3

The I/O register is divided among the four timers on a byte boundary with only two bits being utilized by each timer (See Figures 10.8, 10.9, 10.10, and 10.11). The bits are used to place data on or get data off an external pin.

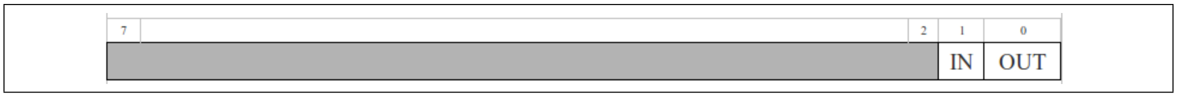


Figure 10.8: MiniAT Timer I/O 0

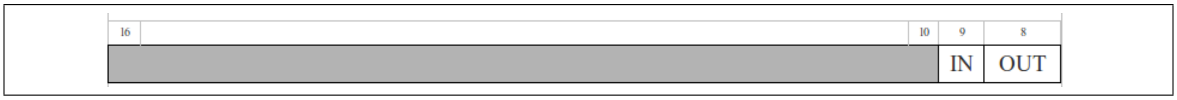


Figure 10.9: MiniAT Timer I/O 1

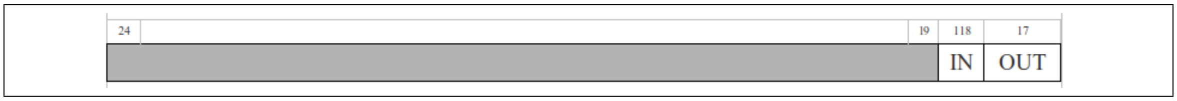


Figure 10.10: MiniAT Timer I/O 2

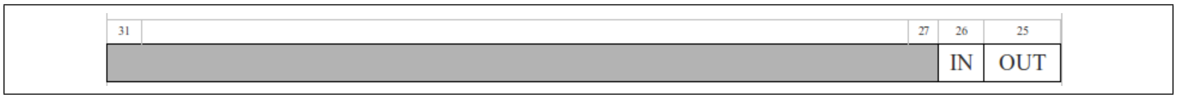


Figure 10.11: MiniAT Timer I/O 3

The supported timer modes are listed below:

- Time - The "time" mode of operation: if $TimerXCount >= TimerXCompare$, then IFRA→Timer X and Timer I/O→TX_O are set to 1; the timer is reset, and Timer Control→Enable is reset to 0; the pre-scalar modulus is reset, and Timer X Count is reset to 0.

- Counter - This mode operates the same as time mode, except Timer X Count increments when Timer I/O→TX_I transitions from 0 to 1.

- Toggle - This mode operates the same as timer mode, except Timer I/O→TX_O is toggled when $TimerXCount >= TimerXCompare$, and the timer is not disabled.

- PWM - In the pulse wave modulation mode of operation, this pulse modulates a waveform on Timer I/O→TX_O based on a 16 bit frequency, stored in the high wyde of Timer X Compare(bits 31 to 16), and 16 bit duty, which is stored in the low wyde of Timer X Compare (bits 15 to 0).

Chapter 11
Software

The MiniAT is more than a new architecture design. The project is comprised of multiple software packages, which includes a simulator, assembler, peripherals, and visualizers.

11.1 Simulator

The simulator is the core of the MiniAT project. It is an implementation of the MiniAT architecture. This includes design characteristic discribed in the thesis, including the instruction set, memory map, pipeline, interrupt handler, etc. The simulator was implemented in “C” as a library and only requires the ‘C’ standard library to be built. The library can be distributed in source code or pre-compiled form. It has been tested on Ubuntu, OSX and Windows. The MiniAT is currently licenced under the GNU public license (GPL) but is subject to change without notice. The only header file required to work with the library is ‘miniat.h’. The interface exposed by the library consists of a few functions (See Figure 11.1) and a single structure (See Figure 11.2). All the types associated with the simulator is defined in the ‘miniat_types.h’ file (this header file does not need to be directly included for use).

```
/* functions */


/* miniat.h */
void miniat_new (miniat *m, FILE *bin_file, char *log_file_name);
void miniat_free (miniat *m);
void miniat_reset(miniat *m);
void miniat_clock (miniat *m);


/* miniat_pins.h */
void m_pins_bus_set(miniat *m, miniat_bus bus);
void m_pins_bus_get(miniat *m, miniat_bus *bus);
void m_pins_set_xint(miniat *m, uwyde data);
void m_pins_get_xint(miniat *m, uwyde *data);
void m_pins_set_timers(miniat *m, ubyte data);
void m_pins_get_timers(miniat *m, ubyte *data);
void m_pins_set_gpio_portA(miniat *m, uwyde data);
void m_pins_get_gpio_portA(miniat *m, uwyde *data);
void m_pins_set_gpio_portB(miniat *m, uwyde data);
void m_pins_get_gpio_portB(miniat *m, uwyde *data);
void m_pins_set_gpio_portC(miniat *m, uwyde data);
void m_pins_get_gpio_portC(miniat *m, uwyde *data);
void m_pins_set_gpio_portD(miniat *m, uwyde data);
void m_pins_get_gpio_portD(miniat *m, uwyde *data);
```

Figure 11.1: The functions given are all that is required to interact with a MiniAT simulator. The pin functions are only required if external interaction is desired.

The *miniat_new* function must be called in order to initialize the simulator. The first parameter is a pointer to an instance of a miniat (m). The second parameter is a pointer to a file (*bin_file*) which contains a binary image to be loaded into the simulators flash memory. The third parameter is the file name (*log_file_name*) for the log file to be used. If *NULL* is passed as the *log_file_name*, then the log defaults to *stderr*.

The *miniat_free* function is called when the simulator is no longer needed. This function frees all of the internal structures and only requires a pointer to the miniat(m) to be passed to it.

The *miniat_reset* function can be called to restart the simulator. It will clear out all registers and reset all memory areas. The only parameter required is a pointer to an instance of the miniat(m).

The *miniat_clock* function needs to be called in order for the simulator to perform a clock cycle. Usually this function will be called in a loop to simulate multiple clock cycles being given to the simulator. The only parameter required is a pointer to an instance of the miniat(m).

The *miniat_pins_** functions allow for access to the exposed virtual pins on the simulator. The first parameter required is a pointer to an instance of the miniat(m). If the function is a *get* then the second parameter is a pointer to the bus or a uwyde. If the function is a *set*, then the second parameter is the bus or a uwyde.

```
/* structures */
typedef struct miniat miniat;


struct miniat {
uword mem[M_MEM_SIZE];
uword reg[M_NUM_REGISTERS];
miniat_interrupts interrupts;
miniat_timers timers;
miniat_pipeline pipeline;
miniat_bus bus;
miniat_csr *csr;
uword ticks;
};
```

Figure 11.2: This is the MiniAT structure. It is all that is required to create and uses the provided functions for interaction. Knowledge of the internal structures is not necessary to use the simulator. All of the sub structures are defined in ‘miniat_types.h’ and are useful for inspection of the simulators inter-workings.

The *miniat* structure should not be accessed directly by the environment or peripherals. This structure should only be used for debugging purposes and *not* in place of the pins interface for interaction.

The *uword mem* array is the memory of the simulator (See Figure 4.2.2). The *uword reg* array is the register file. The *miniat_interrupts* structure is used to hold internal data for the priority based interrupt algorithm (see section 7). The *miniat_timers* structure holds pointers into the memory map for the watchdog and general purpose timers and exists to make access easier. The *miniat_pipeline* structure contains all of the latches for the pipeline stages, the PCQ, stalls, and data hazard queues. The *miniat_bus* structure holds state information about the bus. The *miniat_csr* structure contains pointers into the memory map for easy access to the control status registers. The *uword ticks* holds the number of clock cycles starting from the last time either *miniat_new* or *miniat_reset* was called.

See Figure 11.3 for example usage of the simulator.

```c
#include "miniat.h"
#include "peripheral.h"
int main(int argc, char *argv[]) {
        miniat m;
        peripheral p;
        FILE *fp;
        fp = fopen("binary_file.bin", "r");
        miniat_new(&m, fp);
        peripheral_new(&p);
        while(!exit_event) {
                /* CONNECT PINS */
                miniat_clock(&m);
                peripheral_clock(&p);
        }
        miniat_free(&m);
        peripheral_free(&p);
        return 0;
}
```

Figure 11.3: This example was developed in C.

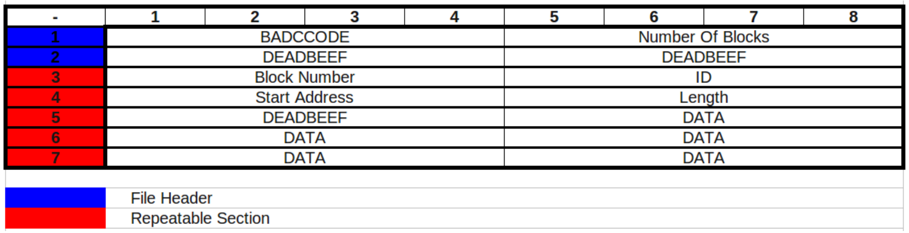
11.2 Assembler

SHASM is the current assembler for the MiniAT. The assembler requires the flex and lemon packages. SHASM has been built and tested on Ubuntu. SHASM is very basic when compared to other assemblers. It provides functionality to comment, define constants, labels. All instructions must explicitly define all encoded pieces. Pseudo instructions are also not supported as of the time of this writing. Even though only basic features are included, it speeds up development time drastically over writing binary files by hand. Example code is given to illustrate the basic format of a assembly file (See Figure 11.4.

```
;this is the starting address for the section
.address 0x2000
;this is a label
!main
;instructions
ADD r1, r0, r0, 5
BRAE {F} r0, r0, r0, !main
ADD r1, r1, r0, 2
ADD r0, r0, r0, 0
ADD r0, r0, r0, 0
```

Figure 11.4: The example shows a program that places a 5 in r1 and then branches to !main. Keep in mind that the branch instruction has a 3 cycle delay slot and will also perform the 3 trailing instructions.

The binary file produced must follow a very specific format to be parsed correctly by the *miniat_new* function. The file is organized in little endian byte ordering. Figure 11.5 shows the required file layout.



-	1	2	3	4	5	6	7	8
🟦 1	BADCCODE				Number Of Blocks			
🟦 2	DEADBEEF				DEADBEEF			
🟥 3	Block Number				ID			
🟥 4	Start Address				Length			
🟥 5	DEADBEEF				DATA			
🟥 6	DATA				DATA			
🟥 7	DATA				DATA			

🟦	File Header
🟥	Repeatable Section

Figure 11.5: The figure shows the required file block format for a binary file that will be loaded by the *miniat_new* function. The rows in blue identify the file header and the rows in red identify the block section. The block section could repeat based on the number of blocks identified in the file header.

The file starts with a 32 bit unique identifier for the miniat format which in HEX is 0xBADCC0DE. Then the number of blocks will be identified in a 32 bit number. The unique identifier and the number of blocks are considered the header for the file. The next 8 bytes end the file header with 0xDEADBEEF. The section following the file header could repeat based on the number of blocks identified in the file header. The block number is a unique identifier for the block and must be greater than 0 and less than the number of blocks. ID identifies a section that the block belongs to. It is currently unused, but is intended for state restoration. Start address is an address to start writing data to. Length is used to identify how much data exists in the block. A 0xDEADBEEF identifier is next and then the data. An example binary file created with SHASM is given below (See Figure 11.6).

```
0000000: dec0 dcba 0100 0000 efbe adde efbe adde  ................
0000010: 0000 0000 0100 0000 0020 0000 0a00 0000  ......... ......
0000020: efbe adde 0000 0100 0500 0000 0000 0016  ................
0000030: 0020 0000 0001 0100 0200 0000 0000 0000  . ..............
0000040: 0000 0000 0000 0000 0000 0000            ............
```

Figure 11.6: This is an example of a binary file produced by SHASM. The dump displayed above was creaded with *xxd*. The file assembled was Figure 11.4. The binary file is presented in little endian as required by the MiniAT simulator.

11.3 Visualizer

SHUI is the current visualizer for the MiniAT. At the time of writing it displays the following data:

- Pipeline
- PCQ
- Data Dependencies
- Structural Dependencies
- Low 5 and High 5 Registers
- Timers
- Bus state
- Ticks

SHUI can be easily integrated into any developed environment with just 1 instance of the *shui* structure and 3 function calls as shown in figure 11.7. The purpose of SHUI is to show the overall state of the microcontroller. This can be extremely useful when debugging an application. SHUI is also useful to help educate people on the interworkings of a pipelined microcontroller.

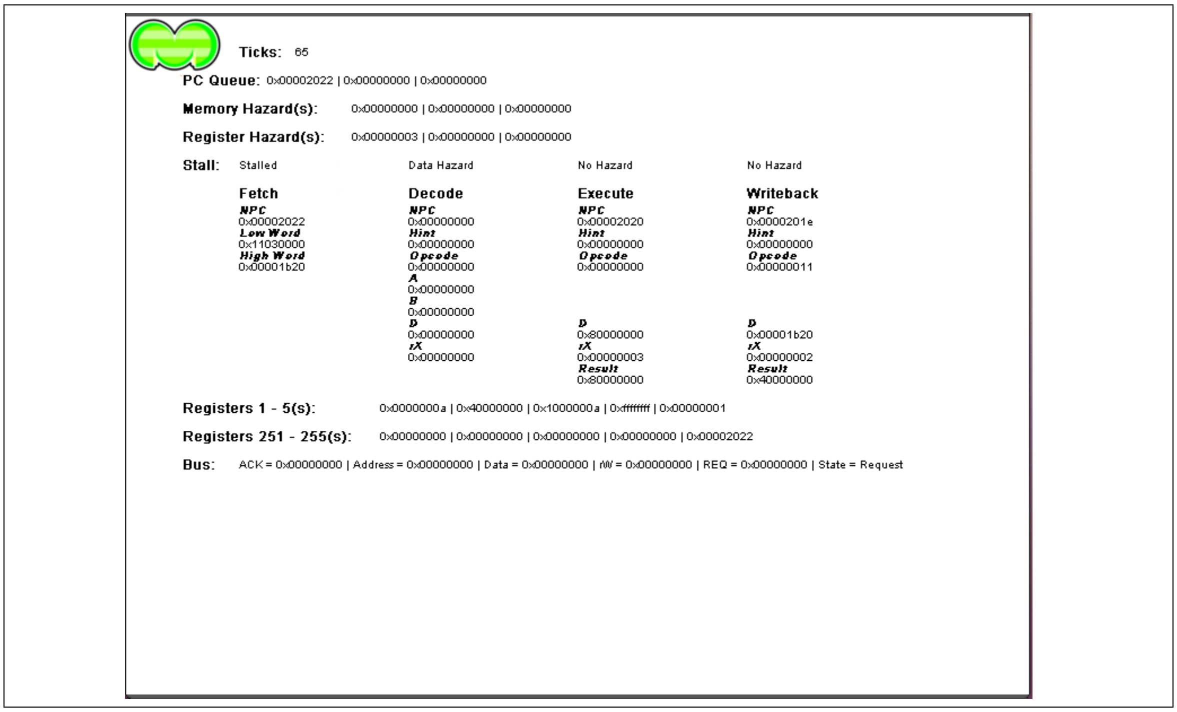
```
#include "miniat.h"
#include "shui.h"

#include <stdbool.h>

int main(int argc, char *argv[]) {
        miniat m;
        shui s;
        FILE *fp;
        int exit_event = 0;
        fopen("binary_file.bin", "r");
        miniat_new(&m, fp);
        shui_new(&s);           //<---------------------SHUI CALL HERE
        while(!exit_event) {
                miniat_clock(&m);
                if(shui_clock(&s)) {       //<---------SHUI CALL HERE
                        exit_event = TRUE;
                }
        }
        miniat_free(&m);
        shui_free(&p);         //<----------------------SHUI CALL HERE
        return 0;
}
```

Figure 11.7: As shown, SHUI only requires 3 lines of active code to be inserted into the environment program.

A screen shot is also presented (See Figure 11.8).



```
Ticks: 65
PC Queue: 0x00002022 | 0x00000000 | 0x00000000
Memory Hazard(s):     0x00000000 | 0x00000000 | 0x00000000
Register Hazard(s):   0x00000003 | 0x00000000 | 0x00000000
Stall:   Stalled         Data Hazard     No Hazard       No Hazard
         Fetch           Decode          Execute         Writeback
         NPC             NPC             NPC             NPC
         0x00002022      0x00000000      0x00002020      0x0000201e
         Low Word        Hint            Hint            Hint
         0x11030000      0x00000000      0x00000000      0x00000000
         High Word       Opcode          Opcode          Opcode
         0x00001b20      0x00000000      0x00000000      0x00000011
                         A
                         0x00000000
                         B
                         0x00000000
                         D               D               D
                         0x00000000      0x80000000      0x00001b20
                         rX              rX              rX
                         0x00000000      0x00000003      0x00000002
                                         Result          Result
                                         0x80000000      0x40000000
Registers 1 - 5(s):       0x0000000a | 0x40000000 | 0x1000000a | 0xffffffff | 0x00000001
Registers 251 - 255(s):   0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00002022
Bus:   ACK = 0x00000000 | Address = 0x00000000 | Data = 0x00000000 | rW = 0x00000000 | REQ = 0x00000000 | State = Request
```

Figure 11.8: This is a screen shot of SHUI showing watchdog.bin executing.

11.4 Development Roles

Now that the simulator is ready for release, development will start to shift to other areas of the project. The authors have identified 3 different development areas/roles:

- Environment Developer
- Peripheral Developer
- Assembly Developer

11.4.1 Environment Developer

The environment developer would be responsible for integrating peripherals with the MiniAT. This could include multiple instances of the architecture and/or peripherals. This category of development does not require extensive knowledge of the internal architecture or peripheral architecture because the interface for each will be exposed as pins and a clock cycle function. With the limited implementation overhead, the developer can focus on the environment being created. Some examples of the environment could be anything from a basic connection between a terminal and keyboard, or as complex as hundreds of simulators acting as nodes on a virtual network.

11.4.2 Peripheral Developer

The peripheral programmer focuses on the creation of new and interesting peripherals to integrate with the simulator. Peripherals could be virtual recreation of physical equipment such as:

- Accelerometer
- GPS
- Proximity Sensor
- Keyboard
- Terminal
- Speakers
- Printer
- Network Card

The other approach a peripheral programmer could take would be to extend a system device driver to the MiniAT pin interface and allow the simulator to actually control real hardware. Either approach allows for the assembly programmer to work with virtual or real hardware without the overhead of operating system development.

In either case the interface exposed by the peripheral is just pins and can be easily connected to the simulator by the environment programmer.

11.4.3 Assembly Developer

The task of the assembly programmer is to create applications that will be loaded by the MiniAT. This developer would have to understand what environment the simulator was placed in and how to interact with connected peripherals, as well as how to interact with the MiniAT itself.

Chapter 12

Future Work

The MiniAT explored a wide range of architectural concepts in the creation of the first architecture. There are many areas of interest left to pursue. Some areas of interest to the project are listed below.

1. Instruction set
2. Analyzing the ISA
3. Bus
4. Support for operating systems
5. Very Large Instruction Word Architecture

12.1 Instruction Set

The Very Similair Instruction set Architecture design uses a memory intensive 64-bit format for all instructions. This allows for easy use of immediate values; however, immediate values are not typically used in general software. Most values are acquired from the environment and configuration files. The VSIA single encoding has a reserved bit that could be utilized to address this issue. In a system similar to ARM's THUMB instructions[22], the reserved bit could be used to switch the machine into another instruction mode, where the instructions have a smaller bit width than the current ISA.

12.2 Analyzing the ISA

Since VSIA uses a single encoding, a comparison of decode circuitry complexity for various ISAs would be of interest. So does the current VSIA design reduce decode circuitry complexity?

12.3 Bus

The MiniAT bus has room for improvement as well, and adding support for the following would be a welcome enhancement:

- Multi-master, multi-slave support
- Ability for the slave to initiate requests with the master
- Inclusion of a bus interrupt line for explicit bus driven I/O through a common device independent interface.

12.4 Pedagogy

Since one of the major goals of the project was pedagogy, responses from faculty and students that use the simulator would be very beneficial. Allowing the us, or someone else, to compare the strengths and weaknesses of the MiniAT design across multiple application environments would provide crucial direction for another simulator. Users wishing to provide feedback can contact the MiniAT team through the website `www.miniat.org`. Current plans are to have the website running by May 2011.

12.5 Support for OSs

The MiniAT architecture is an exceptional embedded microcontroller for very specific environments. However, it lacks features for supporting a full operating system. Endeavors into architectural necessities for secure operating system development is another possible avenue of future MiniAT architecture design.

12.6 VLIW

Another architectural concept to be explored is VLIW. Although the paper covered the Itanium[22], the aspects of VLIW were not examined. Another potential MiniAT architecture could be a VLIW architecture. Work in this area could be pursued.

Chapter 13

Conclusion

The MiniAT has blended many existing ideas into a single architecture. The project has also put a unique twist on many of these features and completely introduced new concepts in other areas. Although this paper closes the definition of the first architecture produced, it does not end the project itself, there is still a plethora of future endeavors; from further research and development with the current design to new architectures. One of the goals of this project was pedagogy, and hopefully other students can benefit from our work currently done (as the author's did), and enhance the work for their own purposes. The authors look forward to the simulator and tools built by this project being used in the classroom as well as feed back from those who use it.

Bibliography

[1] `http://necrobones.com/atrobots/`.

[2] Ola Ågren. Teaching computer concepts using virtual machines. *SIGCSE Bulletin*, 31(2):84–85, 1999.

[3] Ola Ågren. Virtual machines as an aid in teaching computer concepts. In *WCAE '00: Proceedings of the 2000 workshop on Computer architecture education*, page 14, 2000.

[4] III B. Lewis Barnett. A visual simulator for a simple machine and assembly language. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 233–237, 1995.

[5] Milos Becvar, Alois Pluhacek, and Jiri Danacek. Dop - a simple processor core for educational purposes. In *Programmable Devices and Systems 2003) : a proceedings volume from the 6th IFAC Workshop*, pages 208–213, 2003.

[6] Milos Becvar, Alois Pluhacek, and Jiri Danecek. Dop: a cpu core for teaching basics of computer architecture. In *WCAE '03: Proceedings of the 2003 workshop on computer architecture education*, page 4, 2003.

[7] Ewa Z. Bem. A case for teaching computer architecture. In *ACE '03: Proceedings of the fifth Australasian conference on Computing education*, pages 1–7, 2003.

[8] Ewa Z. Bem and Luke Petelczyc. Minimips: a simulation project for the computer architecture laboratory. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 64–68, 2003.

[9] Grant Braught. Teaching empirical skills and concepts in computer science using random walks. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 41–45, 2005.

[10] Grant Braught, Craig S. Miller, and David Reed. Core empirical concepts and skills for computer science. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 245–249, 2004.

[11] Grant Braught and David Reed. The knob & switch computer: A computer architecture simulator for introductory computer science. *Journal on Educational Resources in Computing*, 1(4):31–45, 2001.

[12] Grant Braught and David Reed. Disequilibration for teaching the scientific method in computer science. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 106–110, 2002.

[13] Mats Brorsson. Mipsit: a simulation and development environment using animation for computer architecture education. In *WCAE '02: Proceedings of the 2002 workshop on Computer architecture education*, page 12, 2002.

[14] Kim Buckner. A non-traditonal approach to an assembly language course. *Journal of Computing Sciences in Colleges*, 22(1):179–186, 2006.

[15] Jan Van Campenhout, Peter Verplaetse, and Henk Neefs. Escape: environment for the simulation of computer architectures for the purpose of education. In *WCAE '98: Proceedings of the 1998 workshop on Computer architecture education*, page 9, 1998.

[16] Ajay Chopra. `http://www.atarimuseum.com/articles/pbi-1.html`, 1983.

[17] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, 1994.

[18] Marc L. Corliss and Robert Hendry. Larc: a little architecture for the classroom. *Journal of Computing in Small Colleges*, 24(6):15–20, 2009.

[19] Marius Cornea-Hasegan. Ia-64 floating-point operations and the ieee standard for binary floating-point arithmetic. *Intel Technology Journal*, 1999.

[20] Thad Crews and Uta Ziegler. The flowchart interpreter for introductory programming courses. In *FIE '98: Proceedings of the 1998 28th Annual Frontiers in Education Conference*, pages 307–312, 1998.

[21] Shirley Crossley, Hugh Osborne, and William Yurcik. How computers really work: a children's guide. In *WCAE '02: Proceedings of the 2002 workshop on Computer architecture education*, page 14, 2002.

[22] Sivarama P. Dandamudi. *Guide to RISC Processors for programmers and engineers.* Springer Science+Media Business Inc, 2005.

[23] Rick Decker and Stuart Hirshfield. The pippin machine: simulations of language processing. *Journal on Educational Resources in Computing*, 1(4):4–17, 2001.

[24] Barry Donahue. Using assembly language to teach concepts in the introductory course. In *SIGCSE '88: Proceedings of the nineteenth SIGCSE technical symposium on Computer science education*, pages 158–162, 1988.

[25] B. Dugan and J. Zahorjan. The sloop isa and the smok toolkit. *Journal on Educational Resources in Computing*, 2(1):49–71, 2002.

[26] Roger Duke, Eric Salzman, Jay Burmeister, Josiah Poon, and Leesa Murray. Teaching programming to beginners - choosing the language is just the first step. In *ACSE '00: Proceedings of the Australasian conference on Computing education*, pages 79–86, 2000.

[27] Dan Ellard, Penelope Ellard, James Megquier, and J. Bradely Chen. Ant architecture: an architecture for cs1. In *WCAE-4 '98: Proceedings of the 1998 workshop on Computer architecture education*, page 3, 1998.

[28] Daniel Ellard, David Holland, Nicholas Murphy, and Margo Seltzer. On the design of a new cpu architecture for pedagogical purposes. In *WCAE '02: Proceedings of the 2002 workshop on Computer architecture education*, page 6, 2002.

[29] Stuart Garner. Learning resources and tools to aid novices learn programming. In *IS '03: Proceedings of the Information Science and Information Technology Education Joint Conference*, pages 213–222, 2003.

[30] Alessio Gaspar, Sarah Langevin, William D. Armitage, and Matt Rideout. March of the (virtual) machines: past, present, and future milestones in the adoption of virtualization in computing education. *Journal of Computing Sciences in Colleges*, 23(5):123–132, 2008.

[31] Elena Giannotti. Algorithm animator: a tool for programming learning. In *SIGCSE '87: Proceedings of the eighteenth SIGCSE technical symposium on Computer science education*, pages 308–314, 1987.

[32] Simon Gray, Caroline St. Clair, Richard James, and Jerry Mead. Suggestions for graduated exposure to programming concepts using fading worked examples. In *ICER '07: Proceedings of the third international workshop on Computing education research*, pages 99–110, 2007.

[33] Harvard Computer Science Group. The ant-32 architecture revision 3.1.0b. In *Harvard University School of Engineering and Applied Sciences (SEAS)*, page 32, 2002.

[34] John L. Hennessy. *Computer architecture: a quantitative approach.* Denise E.M. Penrose, 2007.

[35] David Hinkle, David Kortenkamp, and David Miller. The 1995 robot competition and exhibition. *AI Magazine*, 17:31–45, 1996.

[36] Michael S. Horn. Tangible programming with quetzal: Opportunities for education. Master's thesis, Tufts University, 2006.

[37] Michael S. Horn and Robert J. K. Jacob. Tangible programming in the classroom: a practical approach. In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 869–874, 2006.

[38] Michael S. Horn and Robert J. K. Jacob. Designing tangible programming languages for classroom use. In *TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 159–162, 2007.

[39] Michael S. Horn and Robert J. K. Jacob. Tangible programming in the classroom with tern. In *CHI '07: CHI '07 extended abstracts on Human factors in computing systems*, pages 1965–1970, 2007.

[40] Scott B. Hunter. Enhancing pedagogy via ebay: some assembly (language) required. *Journal of Computing Sciences in Colleges*, 18(5):13–22, 2003.

[41] Scott B. Hunter. Teaching assembly language without using (as much) assembly language. *Journal of Computing Sciences in Colleges*, 20(5):68–78, 2005.

[42] Kosuke Imamura. Assembly language is more than a teaching tool. *Journal of Computing Sciences in Colleges*, 20(2):49–54, 2004.

[43] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, 1997.

[44] Lubomir Ivanov and John S. Mallozzi. A hardware/software simulator to unify courses in the computer science curriculum. *Journal of Computing Sciences in Colleges*, 19(5):238–248, 2004.

[45] Sahnny Johnson and Fritz Ruehr. An integrated course in architecture and compilers. *Journal of Computing Sciences in Colleges*, 21(1):191–198, 2005.

[46] Russell Kao, 1986.

[47] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, 2005.

[48] Caitlin Kelleher and Randy Pausch. Lessons learned from designing a programming system to support middle school girls creating animated stories. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 165–172, 2006.

[49] James P Kelsh and John C Hansen. A simple virtual machine. *SIGSMALL/PC Notes*, 13(1):11–15, 1987.

[50] J. M. Kerridge and N. Willis. A simulator for teaching computer architecture. *ACM SIGCSE Bulletin*, 12(2):65–71, 1980.

[51] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. The simcore/alpha functional simulator. In *WCAE '04: Proceedings of the 2004 workshop on Computer architecture education*, page 24, 2004.

[52] Donald Knuth. The art of computer programming: Fascicle 1. To be published in an upcoming edition of TAOFCP, 2004.

[53] Jaejin Lee, Junghyun Kim, Choonki Jang, Seungkyun Kim, Bernhard Egger, Kwangsub Kim, and SangYong Han. Facsim: a fast and cycle-accurate architecture simulator for embedded systems. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 89–100, 2008.

[54] James R. Leonard. Using a software engineering approach to cs1: a comparative study of student performance. *SIGCSE Bulletin*, 23(4):23–26, 1991.

[55] S. P. Maj, D. Veal, and P. Charlesworth. Is computer technology taught upside down? In *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education*, pages 140–143, 2000.

[56] Naraig Manjikian. Enhancements and applications of the simplescalar simulator for undergraduate and graduate computer architecture education. In *WCAE '00: Proceedings of the 2000 workshop on Computer architecture education*, page 8, 2000.

[57] Fred G. Martin. Integrating hardware experiences into a computer architecture core course. *Journal of Computing Sciences in Colleges*, 21(6):39–52, 2006.

[58] Bernd Mathiske. The maxine virtual machine and inspector. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 739–740, 2008.

[59] Richard E. Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys*, 13(1):121–141, 1981.

[60] Richard E. Mayer, Jennifer L. Dyck, and William Vilberg. Learning to program and learning to think: what's the connection? *Communications of the ACM*, 29(7):605–610, 1986.

[61] Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline St. Clair, and Lynda Thomas. A cognitive approach to identifying measurable milestones for programming skill acquisition. In *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 182–194, 2006.

[62] Filippo Menczer and Alberto Maria Segre. Oamulator: a teaching resource to introduce computer architecture concepts. *Journal on Educational Resources in Computing*, 1(4):18–30, 2001.

[63] David Miller, Illah Nourbakhsh, and Roland Siegwart. *Robots for education*, chapter 55, pages 1283–1301. Springer, 2008.

[64] David P. Miller and Cathryne Stein. Oh, that's what you use π for! In *Proceedings of the AIAA SARTC Robotics Workshop*, July 1998.

[65] David P. Miller and Charles Winton. Botball kit for teaching engineering computing. In *In Proceedings of the ASEE National Conference. ASEE*, 2004.

[66] Wai Sum Mong and Jianwen Zhu. A retargetable micro-architecture simulator. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 752–757, 2003.

[67] Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the effectiveness of a new instructional approach. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 75–79, 2004.

[68] J. C. Moure, Dolores I. Rexachs, and Emilio Luque. The kscalar simulator. *Journal on Educational Resources in Computing*, 2(1):73–116, 2002.

[69] Mark Newsome, Cherri M. Pancake, and Christopher Ward. Visual execution of assembly language programs. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, pages 38–43, 1993.

[70] B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic. A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *IEEE*, 52(4):449 –458, 2009.

[71] Cindy Norris and James Wilkes. Yess: a y86 pipelined processor simulator. In *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*, pages 150–155, 2007.

[72] Linda Null and Julia Lobur. Mariesim: The marie computer simulator. *Journal on Educational Resources in Computing*, 3(2):1, 2003.

[73] Hugh Osborne. The postroom computer. *Journal on Educational Resources in Computing*, 1(4):81–110, 2001.

[74] Hugh Osborne, Shirley Crossley, Jiří Mencák, and William Yurcik. Pectopah: promoting education in computer technology using an open-ended pedagogically adaptable hierarchy. In *WCAE '02: Proceedings of the 2002 workshop on Computer architecture education*, page 19, 2002.

[75] Seymour Papert. A computer laboratory for elementary schools: Logo memo 1. Technical report, Massachusetts Institute of Technology, 1971.

[76] Enric Pastor, Fermín Sánchez, and Anna M. del Corral. A rudimentary machine: experiences in the design of a pedagogic computer. In *WCAE '98: Proceedings of the 1998 workshop on Computer architecture education*, page 7, 1998.

[77] Robert A. Pilgrim. Design and construction of the very simple computer (vsc): a laboratory project for an undergraduate computer architecture course. In *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, pages 151–154, 1993.

[78] Kris D. Powers. Breadth-also: a rationale and implementation. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 243–247, 2003.

[79] Kris D. Powers. Teaching computer architecture in introductory computing: why? and how? In *ACE '04: Proceedings of the sixth conference on Australasian computing education*, pages 255–260, 2004.

[80] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: a review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[81] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, 2004.

[82] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel and Distributed Technology*, 3:34–43, 1995.

[83] Greg W. Scragg. Most computer organization courses are built upside down. In *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 341–346, 1991.

[84] Phillips Semiconductors. The i2c-bus specification. 2001.

[85] Dale Skrien. Cpu sim 3.1: A tool for simulating computer architectures for computer organization classes. *Journal on Educational Resources in Computing*, 1(4):46–59, 2001.

[86] Dale Skrien and John Hosack. A multilevel simulator at the register transfer level for use in an introductory machine organization class. In *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 347–351, 1991.

[87] Timothy D. Stanley, Thanh Quach Xuan, Leslie Fife, and Don Colton. Simple eight bit, emulated computers for illustrating computer architecture concepts and providing a starting point for student designs. In *ACE '07: Proceedings of the ninth Australasian conference on Computing education*, pages 141–146, 2007.

[88] Timothy Daryl Stanley and Mu Wang. An emulated computer with assembler for teaching undergraduate computer architecture. In *WCAE '05: Proceedings of the 2005 workshop on Computer architecture education*, page 7, 2005.

[89] Jeffrey A. Stone. Using a machine language simulator to teach cs1 concepts. In *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 43–45, 2006.

[90] Dominick Sweetman. *See MIPS Run.* Denise E.M Penrose, 2 edition, 2007.

[91] A. Szekacs, T. Szakall, and Z. Hegykozi. Realising the spi communication in a multiprocessor system. pages 213–216, aug. 2007.

[92] Soe Than. Use of a simulator and an assembler in teaching input-output processing and interrupt handling. *Journal of Computing Sciences in Colleges*, 22(4):75–81, 2007.

[93] L. S. K. Udugama and Janath C Geeganage. Students' experimental processor: a processor integrated with different types of architectures for educational purposes. In *WCAE '06: Proceedings of the 2006 workshop on Computer architecture education*, page 7, 2006.

[94] Timothy Urness. Teaching computer organization/architecture by building a computer. In *WCAE '07: Proceedings of the 2007 workshop on Computer architecture education*, pages 72–76, 2007.

[95] Brian G. VanBuren and Muhammad Shaaban. Microtiger: a graphical microcode simulator with a reconfigurable datapath. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 283–287, 2007.

[96] Kenneth Vollmar and Pete Sanderson. Mars: an education-oriented mips assembly language simulator. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 239–243, 2006.

[97] Gabriel A. Wainer, Sergio Daicz, Luis F. De Simoni, and Demian Wassermann. Using the alfa-1 simulated processor for educational purposes. *Journal on Educational Resources in Computing*, 1(4):111–151, 2001.

[98] Gregory S. Wolffe, William Yurcik, Hugh Osborne, and Mark A. Holliday. Teaching computer organization/architecture with limited resources using simulators. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 176–180, 2002.

[99] Cecile Yehezkel, William Yurcik, Murray Pearson, and Dean Armstrong. Three simulator tools for teaching computer architecture: Little man computer, and rtlsim. *Journal on Educational Resources in Computing*, 1(4):60–80, 2001.

[100] William Tsun yuk Hsu. Experiences integrating research tools and projects into computer architecture courses. In *WCAE '00: Proceedings of the 2000 workshop on Computer architecture education*, page 5, 2000.

[101] William Yurcik and Edward F. Gehringer. A survey of web resources for teaching computer architecture. In *WCAE '02: Proceedings of the 2002 workshop on Computer architecture education*, page 23, 2002.

[102] William Yurcik and Hugh Osborne. A crowd of little man computers: visual computer simulator teaching tools. In *WSC '01: Proceedings of the 33nd conference on Winter simulation*, pages 1632–1639. IEEE Computer Society, 2001.

[103] Yinong Zhang and III George B. Adams. An interactive, visual simulator for the dlx pipeline. In *WCAE-3 '97: Proceedings of the 1997 workshop on Computer architecture education*, page 2, 1995.

[104] Craig Zilles. Spimbot: an engaging, problem-based approach to teaching assembly language programming. In *WCAE '05: Proceedings of the 2005 workshop on Computer architecture education*, page 4, 2005.

Appendices

Appendix A
Appendix

A.1 Acronyms

ACK - Acknowledge
ALAT - Advanced Load Address Table
ALU - Arithmetic Logic Unit
ARM - Advanced RISC Machine
ASM - Assembly
CSR - Control Status Register
DE - Decode
EPIC - Explicit Parallel Instruction Computing
EX - Execute
FI - Flush(F) pipeline on write to the IER(I) register
GPIO - General Purpose Input Output
GUI - Graphical User Interface
IE - Interrupts Enabled
IER - Interrupt Enable Register
IF - Instruction Fetch
IFR - Interrupt Flag Register, this register is divide into two registers, a low(A) and a high(B)
I/O - Input/Output
IP - Instruction Pointer
ISA - Instruction Set Architecture
MiniAT - Mini Advanced Technology
MMU - Memory Management Unit
NaT - Not a Thing
NOP - No Operation Performed
NPC - Next Program Counter
OS - Operating System
PC - Program Counter
PCQ - Program Counter Queue
PWM - Pulse Wave Modulation
RAM - Random Access Memory that is volatile.
REQ - Request
RISC - Reduced Instruction Set Computer
ROM - Read Only Memory that is non-volatile.
RSP - Register Stack Pointer
r/W - Read / Write **SE** - Saturation Enabled
SI - Flush(F) pipeline on write to the System Register(S)

SIMD - Single Instruction Multiple Data
SP - Stack Pointer
uAR - Micro Code Address Register
UI - User Interface
VLIW - Very Long Instruction Word
VSIA - Very Similar Instruction Architecture
WB - Writeback

A.2 Data Sizes

byte - 8 bits
wyde - half-word
uwyde - unsigned half-word
word - largest possible address size for a particular architecture
double word - twice the size of a word