FEESparc

Microprocessor

Graduation Project 2013
Faculty of Electronic Engineering
Menufyia University
Dr.Gamal Mahrous

Contents

1 ARCHITECTURE ……..……
1.1introduction ……..…….
1.2 features ……..……...…
1.3 glossary ……..…….....
1.4 registers ……...…….
1.5 instructions ……..……

2 Microarchitecture …….....
2.1 introduction ……..…..…..
2.1.1architectural states and instruction set …….....
2.1.2 design process ……
2.1.3 mips microarchitecture ……...….
2.2 performance analysis …….
2.3 single cycle processor ……
2.3.1 single cycle data path ……
2.3.2 single cycle control…….....
2.3.3 more instructions ……..
2.3.4 performance analysis ……...……

3 Coding ……

4 References … ……..

Overview

This book is a brief description of the concept an implementation of the graduation project FEESparc Microprocessor, designed by senior students in the Faculty of Electronic Engineering at Menoufyia University. This microprocessor was implemented on FPGA kit.

The project team is :

-Ahmed Ghoneim Elaiat

Department of Industrial Electronics and Control

-Salahuddin Ashraf Abdulrahim

Department of Computer Science and Engineering

-Hassan Adel Eljezeiry

Department of Computer Science and Engineering

Acknowledgment

Many thanks to all who helped to let this project come to light, Dr.Gamal Attiya, Dr.Amr Wassal, and a special thanks to the authors of the great book "Digital Design and Computer Architecture"; Dr.David Harris and Dr.Sarah Harris

CHAPTER ONE

ARCHITECTURE

1.1 Introduction

FEESPARCis a 32-bit single cycle Microprocessor which is based on SPARC Architecture.

SPARC stands for Scalable Processor ARChitecture which is designed by Berkeley University.

."RISC" SPARC derived from Reduced Instruction Set Computer

1.2 Features

- 32-bit address space.
- Few and simple instruction formats — All instructions are 32 bits wide, and are aligned on 32-bit boundaries in memory
- There are only three basic instruction formats, and they feature uniform placement of opcode and register address fields. Only load and store instructions access memory and I/O.
- Few addressing modes — A memory address is given by either "register + register" or "register + immediate."
- Triadic register addresses— Most instructions operate on two register operands (or one register and a constant), and place the result in a third register.
- A large "windowed" register file — At any one instant, a program sees 8 global integer registers plus a 24-register window into a larger register file. The windowed registers can be described as a cache of procedure arguments, local values, and return addresses.

1.3Glossary

The following paragraphs describe some of the most important words and acronyms used :

Current window

The block of 24 *r* registers to which the Current Window Pointer points.

Ignored

Used to describe an instruction field, the contents of which are arbitrary, and which has no effect on the execution of the instruction. The contents of an "ignored" field will continue to be ignored in future versions of the architecture. See also *reserved* and *unused*.Implementation Hardware or software that conforms to all the specifications of an ISA.

Instruction Set Architecture (ISA)

An ISA defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. An ISA does not define clock cycle times, cycles per instruction, data paths, etc.

Next Program Counter (nPC)

Contains the address of the instruction to be executed next (if a trap does not occur).

Privileged

An instruction (or register) that can only be executed (or accessed) when the processor is in supervisor mode (when PSR[S]=1).

Program Counter (PC)

Contains the address of the instruction currently being executed by the IU.

rs1, rs2, rd

Specify the register operands of an instruction. *rs1* and *rs2* are the source registers; *rd* is the destination register

.

Reserved

Used to describe an instruction or register field which is reserved for definition by future versions of the architecture. A reserved field should only be written to zero by software. A reserved register field should read as zero in hardware; software intended to run on future versions of SPARC should not assume that the field will read as zero.

Supervisor Mode

A processor state that is active when the S bit of the PSR is set (PSR[S] = 1).

Trap

A vectored transfer of control to supervisor software through a table whose address is given by a privileged IU register (the Trap Base Register (TBR)).

Unused

Used to describe an instruction field or register field that is not currently defined by the architecture. When read by software, the value of an unused register field is undefined. However, since an unused field could be defined by a future version of the architecture, an unused field should only be written to zero by software.

User Mode

A processor state that is active when the S bit of the PSR is not set (when PSR[S] = 0).

User Application Program

A program executed with the processor in user mode. Also simply called "application program".

1.4 Registers

There are two types of registers: general-purpose or "working" data registers and control/status registers. The IU's **"Integer unit"** general-purpose registers are called ***r*** registers.

1.4.1 IU r Registers

- The IU contains 40 general-purpose 32-bit ***r*** registers.
- They are partitioned into 8 ***global*** registers, plus 2-register ***sets***. A register set is further partitioned into 8 ***in*** registers and 8 ***local*** registers.

Window Addressing

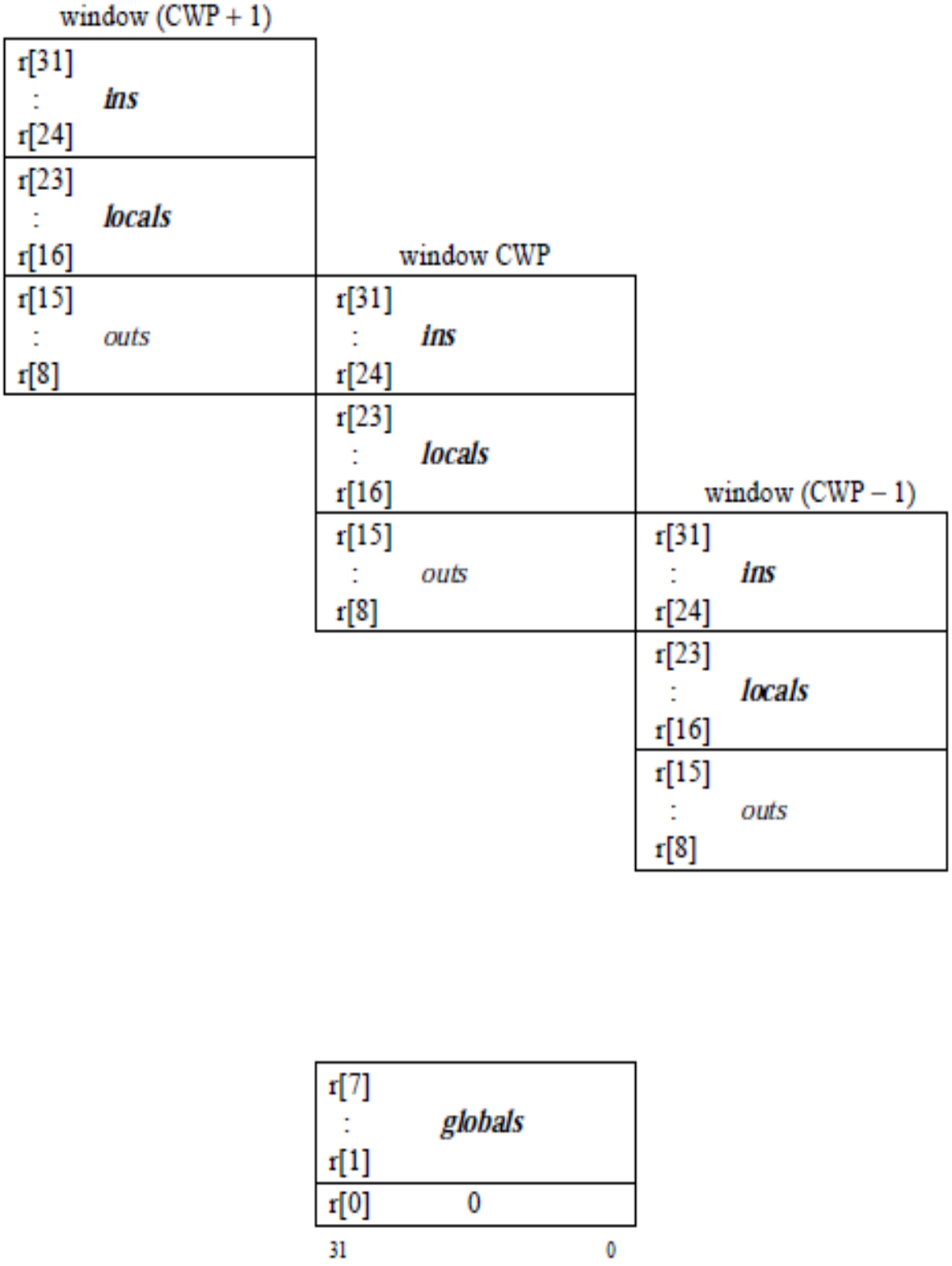
Windowed Register Address	*r* Register Address
in[0] – in[7]	r[24] – r[31]
local[0] – local[7]	r[16] – r[23]
out[0] – out[7]	r[8] – r[15]
global[0] – global[7]	r[0] – r[7]

1.4.1.1Windowed r Registers

At a given time, an instruction can access the 8 *globals* and a 24-register **window** into the *r* registers. A register window comprises the 8 ***in*** and 8 ***local*** registers ofa particular register set, together with the 8 *in* registers of an adjacent register set,which are addressable from the current window as ***out*** registers.

The number of windows or register sets, NWINDOWS, ranges from 2 to 32, depending on the implementation. The total number of *r* registers in a given implementation is 8 (for the *globals*), plus the number of sets □□16 registers/set. Thus, the minimum number of *r* registers is 40 (2 sets), and the maximum number is 520 (32 sets).

The current window into the *r* registers is given by the current window pointer (CWP), a 5-bit counter field in the Processor State Register (PSR). The CWP is incremented by a RESTORE (or RETT) instruction and decremented by a SAVE instruction or a trap. Window overflow and underflow are detected via the window invalid mask (WIM) register, which is controlled by supervisor software.



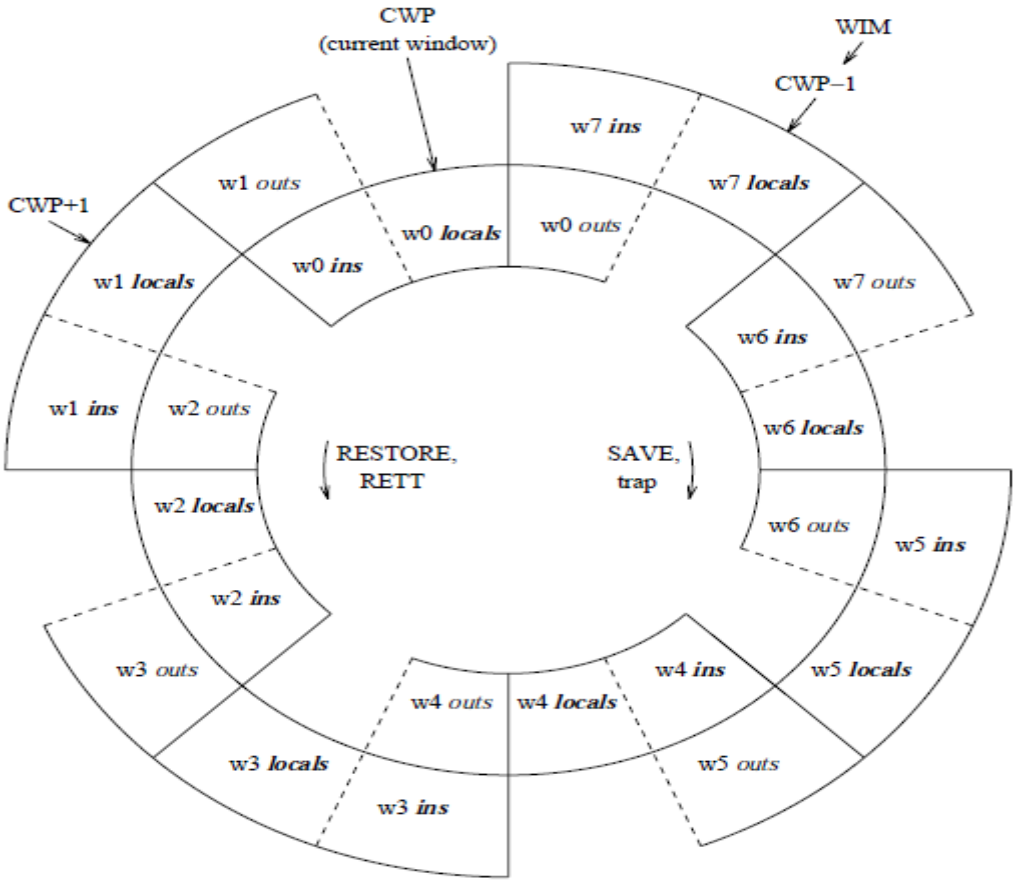
Three Overlapping Windows and the 8 Global Registers

1.4.1.2Overlapping of window

Each window shares its *ins* and *outs* with the two adjacent windows. The *outs* of the CWP−1 window are addressable as the *ins* of the current window, and the *outs* in the current window are the *ins* of the CWP−1 window. The *locals* are unique to each window.

An *r* register with address *o*, where 8 ≤ *o* ≤ 15, refers to exactly the same register as (*o* + 16) does after the CWP is decremented by 1 (modulo NWINDOWS).Likewise, a register with address *i*, where 24 ≤ *i* ≤ 31, refers to exactly the same register as address (*i* − 16) does after the CWP is incremented by 1 (modulo NWINDOWS).

Since CWP arithmetic is performed modulo NWINDOWS, the highest numbered implemented window overlaps with window 0. The *outs* of window 0 are the *ins* of window NWINDOWS−1. Implemented windows must be contiguously numbered from 0 through NWINDOWS−1.



The Windowed r Registers

1.4.2IU Control/Status Registers

The 32-bit IU control/status registers include the Processor State Register (PSR), the Window Invalid Mask register (WIM), the Trap Base Register (TBR), the program counters (PC), and optional, implementation-dependent Ancillary State Registers (ASRs).

1.4.2.1Processor State Register (PSR)

The 32-bit PSR contains various fields that control the processor and hold status information. It can be modified by the SAVE, RESTORE, Ticc, and RETT instructions, and by all instructions that modify the condition codes. The privileged RDPSR and WRPSR instructions read and write the PSR directly.

PSR Fields

impl	*ver*	*icc*	reserved	EC	EF	PIL	S	PS	ET	CWP
31:28	27:24	23:20	19:14	13	12	11:8	7	6	5	4:0

The PSR provides the following fields:

PSR_implementation (impl)

Bits 31 through 28 are hardwired to identify an implementation or class of implementations of the architecture. The hardware should not change this field inresponse to a WRPSR instruction. Together, the PSR.*impl* and PSR.*ver* fieldsdefine a **unique** implementation or class of implementations of the architecture.

PSR_version (ver)

Bits 27 through 24 are implementation-dependent. The *ver* field is either hardwired to identify one or more particular implementations or is a readable and writable state field whose properties are implementation-dependent

PSR_integer_cond_codes (icc)

Bits 23 through 20 are the IU's condition codes. These bits are modified by the arithmetic and logical instructions whose names end with the letters **cc** (e.g.,ANDcc), and by the WRPSR instruction. The Bicc and Ticc instructions cause a transfer of control based on the value of these bits, which are defined as follows:

PSR_negative (n)

Bit 23 indicates whether the 32-bit 2's complement ALU result was negative for the last instruction that modified the *icc* field. 1 = negative, 0 = not negative.

PSR_zero (z)

Bit 22 indicates whether the 32-bit ALU result was zero for the last instruction that modified the *icc* field. 1 = zero, 0 = nonzero.

PSR_overflow (v)

Bit 21 indicates whether the ALU result was within the range of (was representable in) 32-bit 2's complement notation for the last instruction that modified the *icc* field. 1 = overflow, 0 = no overflow.

PSR_carry (c)

Bit 20 indicates whether a 2's complement carry out (or borrow) occurred for the last instruction that modified the *icc* field. Carry is set on addition if there is a carry out of bit 31. Carry is set on subtraction if there is borrow into bit 31. 1 =carry, 0 = no carry.

PSR_reserved

Bits 19 through 14 are reserved. When read by a RDPSR instruction, these bits deliver zeros. For future compatibility, supervisor software should only issue WRPSR instructions with zero values in this field.

PSR_enable_coprocessor (EC)

Bit 13 determines whether the implementation-dependent coprocessor is enabled.If disabled, a coprocessor instruction will trap. 1 = enabled, 0 = disabled. If an implementation does not support a coprocessor in hardware, PSR.EC should always read as 0 and writes to it should be ignored.

PSR_enable_floating-point (EF)

Bit 12 determines whether the FPU is enabled. If disabled, a floating-point instruction will trap. 1 = enabled, 0 = disabled. If an implementation does not support a hardware FPU, PSR.EF should always read as 0 and writes to it should be ignored.

PSR_proc_interrupt_level (PIL)

Bits 11 (the most significant bit) through 8 (the least significant bit) identify the interrupt level above which the processor will accept an interrupt.

PSR_supervisor (S)

Bit 7 determines whether the processor is in supervisor or user mode. 1 = supervisor mode, 0 = user mode.

PSR_previous_supervisor (PS)

Bit 6 contains the value of the S bit at the time of the most recent trap.

PSR_enable_traps (ET)

Bit 5 determines whether traps are enabled. A trap automatically resets ET to 0. When ET=0, an interrupt request is ignored and an exception trap causes the IUto halt execution, which typically results in a reset trap that resumes execution at address 0. 1 = traps enabled, 0 = traps disabled.

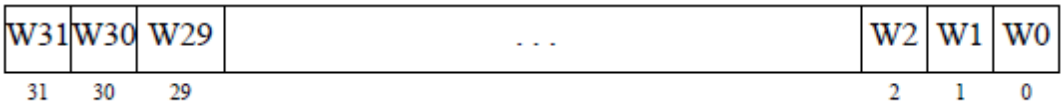
PSR_current_window_pointer (CWP)

Bits 4 (the MSB) through 0 (the LSB) comprise the current window pointer, a counter that identifies the current window into the *r* registers. The hardware decrements the CWP on traps and SAVE instructions, and increments it on RESTORE and RETT instructions (modulo NWINDOWS).

1.4.2.2 Window Invalid Mask Register (WIM)

The Window Invalid Mask register (WIM) is controlled by supervisor software and is used by hardware to determine whether a window overflow or underflow trap is to be generated by a SAVE, RESTORE, or RETT instruction

WIM Fields



There is an active state bit in the WIM for each register set or window in an implementation. WIM[*n*] corresponds to the register set addressed when CWP =*n*.

When a SAVE, RESTORE, or RETT instruction executes, the current value of the CWP is compared against the WIM. If the SAVE, RESTORE, or RETT instruction would cause the CWP to point to an "invalid" register set, that is, one whose corresponding WIM bit equals 1 (WIM[CWP] = 1), a window_overflow or window_underflow trap is caused.

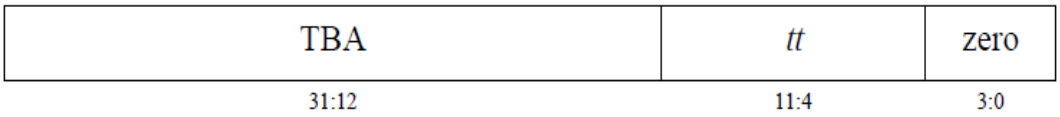
The WIM can be read by the privileged RDWIM instruction and written by the WRWIM instruction. Bits corresponding to unimplemented windows read as zeroes and values written to unimplemented bits are unused. A WRWIM with all bits set to 1, followed by a RDWIM, yields a bit vector in which the implemented windows (and only the implemented windows) are indicated by 1's.

The WIM allows for implementations with up to 32 windows.

1.4.2.3 Trap Base Register (TBR)

The Trap Base Register (TBR) contains three fields that together equal the address to which control is transferred when a trap occurs.

TBR Fields



The TBR provides the following fields:

TBR_trap_base_address (TBA)

Bits 31 through 12 are the trap base address, which is established by supervisor software. It contains the most-significant 20 bits of the trap table address. The TBA field is written by the WRTBR instruction.

TBR_trap_type (tt)

Bits 11 through 4 comprise the trap type (*tt*) field. This 8-bit field is written by the hardware when a trap occurs, and retains its value until the next trap. It provides an offset into the trap table. The WRTBR instruction does not affect the *tt* field.

TBR_zero (0)

Bits 3 through 0 are zeroes. The WRTBR instruction does not affect this field.For future compatibility, supervisor software should only issue a WRTBR instruction with a zero value in this field.

1.4.2.4Ancillary State Registers (ASR)

SPARC provides for up to 31 Ancillary State Registers (ASR's), numbered from 1 to 31.

ASR's numbered 1-15 are reserved for future use by the architecture and should not be referenced by software.

ASR's numbered 16-31 are available for implementation-dependent uses, such as timers, counters, diagnostic registers, self-test registers, and trap-control registers.A particular IU may choose to implement from zero to sixteen of theseASR's. The semantics of accessing any of these ASR's is implementationdependent.Whether a particular Ancillary State Register is privileged or not is implementation-dependent.

An ASR is read and written with the RDASR and WRASR instructions. A read/write ASR instruction is privileged if the accessed register is privileged.

1.5Instructions

Instructions are accessed by the processor from memory and are executed, annulled, or trapped. Instructions are encoded in three 32-bit formats and can be partitioned into four general categories. There are 72 basic instruction operations.

Instructions fall into four basic categories:

1) Load/store

2) Arithmetic/logical/shift

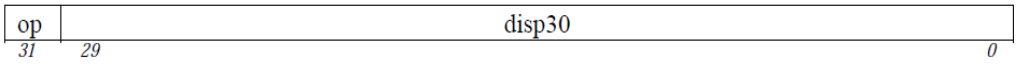
3) Control transfer

4) Read/write control register

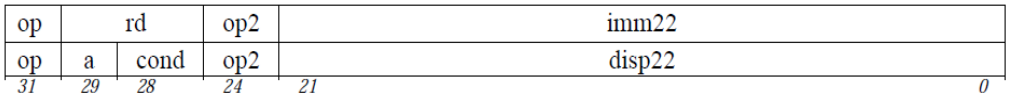
1.5.1 Instruction Formats

Instructions are encoded in three major 32-bit formats.

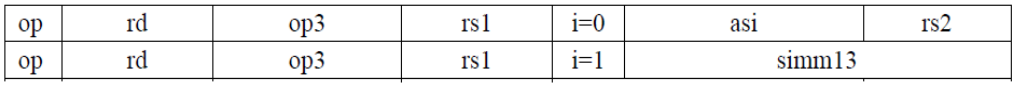
Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



Format 3 (*op* = 2 or 3): Remaining instructions



1.5.2 Instruction Fields

The instruction fields are interpreted as follows:

op and op2

These 2- and 3-bit fields encode the 3 major formats and the format 2 instructions

op Encoding (All Formats)		
Format	*op*	Instructions
1	1	CALL
2	0	Bicc, SETHI
3	3	memory instructions
3 remaining	2	arithmetic, logical, shift, and

op2 Encoding (Format 2)	
op2	*Encoding (Format 2)*
0	UNIMP
1	unimplemented
2	Bicc
3	unimplemented
4	SETHI
5	unimplemented

rd

This 5-bit field is the address of the destination for a load/arithmetic (or store) instruction. For an instruction that read/writes a double (or quad), the least significant one (or two) bits are unused and should be supplied as zero by software.

a

The *a* bit in a branch instruction annuls the execution of the following instruction if the branch is conditional and untaken or if it is unconditional and taken.

cond

This 4-bit field selects the condition code(s) to test for a branch instruction.

imm22

This 22-bit field is a constant that SETHI places in the upper end of a destination register.

disp22 and *disp30*

These 30-bit and 22-bit fields are word-aligned, sign-extended, PC-relative displacements for a call or branch, respectively.

op3

This 6-bit field (together with 1 bit from *op*) encodes the format 3 instructions.

i

The *i* bit selects the second ALU operand for (integer) arithmetic and load/store instructions. If *i* = 0, the operand is r[rs2]. If *i* = 1, the operand is *simm13*, sign-extended from 13 to 32 bits.

asi

This 8-bit field is the address space identifier supplied by a load/store alternate instruction.

rs1

This 5-bit field is the address of the first *r* register source operand. For an instruction that reads a double (or quad), the least significant bit (or 2 bits) are unused and should be supplied as zero by software.

rs2

This 5-bit field is the address of the second *r* register source operand when *i* = 0. For an instruction that reads a double-length (or quad-length) register sequence, the least significant bit (or 2 bits) are unused and should be supplied as zero by software.

simm13

This 13-bit field is a sign-extended 13-bit immediate value used as the second ALU operand for an (integer) arithmetic or load/store instruction when *i* = 1.

1.5.2Instruction Definitions

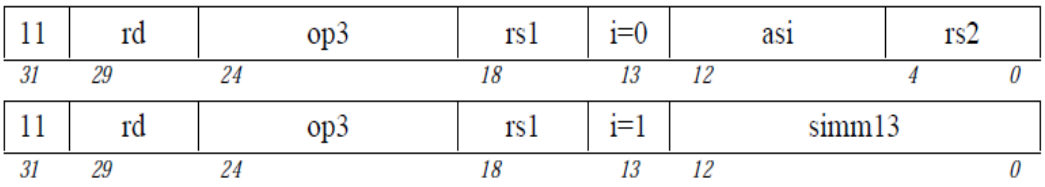
Instruction Set	
Opcode	Name
LD (LDA)	Load Word (from Alternate space)
ST (STA)	Store Word (into Alternate space)
SWAP (SWAPA)	Swap r Register with Memory (in Alternate space)
SETHI	Set High 22 bits of r Register
NOP	No Operation
AND (ANDcc)	And (and modify icc)
ANDN (ANDNcc)	And Not (and modify icc)
OR (ORcc)	Inclusive-Or (and modify icc)
ORN (ORNcc)	Inclusive-Or Not (and modify icc)
XOR (XORcc)	Exclusive-Or (and modify icc)
XNOR (XNORcc)	Exclusive-Nor (and modify icc)
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
ADD (ADDcc)	Add (and modify icc)
SUB (SUBcc)	Subtract (and modify icc)
SAVE	Save caller's window
RESTORE	Restore caller's window

Bicc	Branch on integer condition codes
CALL	Call and Link
JMPL	Jump and Link
RETT	Return from Trap
Ticc	Trap on integer condition codes
RDASR	Read Ancillary State Register
RDPSR	Read Processor State Register
RDWIM	Read Window Invalid Mask Register
RDTBR	Read Trap Base Register
WRASR	Write Ancillary State Register
WRPSR	Write Processor State Register
WRWIM	Write Window Invalid Mask Register
WRTBR	Write Trap Base Register

Load Integer Instruction

opcode	op3	operation
LD	000000	Load Word
LDA Alternate space	010000	Load Word from

Format (3):



Description:

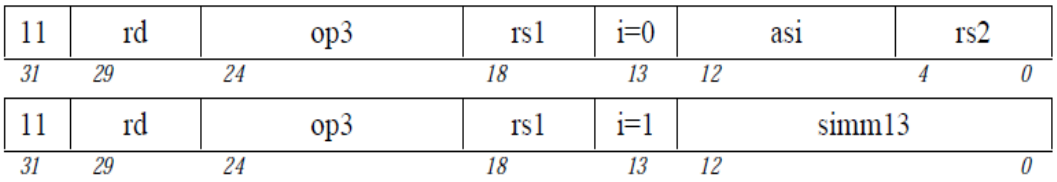
The load integer instructions copy a word from memory into r[*rd*].

The effective address for a load instruction is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one. Instructions that load from an alternate address space contain the address space identifier to be used for the load in the *asi* field, and must contain zero in the *i* field or an illegal_instruction trap will occur. Load instructions that do not load from an alternate address space access either a user data space or system data space, according to the S bit of the PSR.

Store Integer Instruction

opcode	op3	operation
ST	000100	Store Word
STA Alternate space	010100	Store Word into

Format (3):



Description:

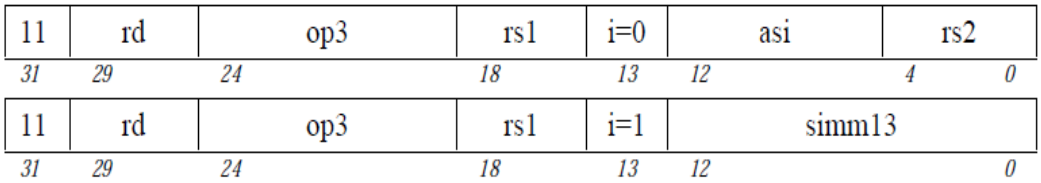
The store integer instructions copy the word from r[*rd*] into memory.

The effective address for a store instruction is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one. Instructions that store to an alternate address space contain the address space identifier to be used for the store in the *asi* field, and must contain zero in the *i* field or an illegal_instruction trap will occur. Store instructions that do not store to an alternate address space access either a user data space or system data space, according to the S bit of the PSR.

SWAP Register with Memory Instruction

opcode	op3	operation
SWAP memory	001111	SWAP register with
SWAPA Alternate space memory	011111	SWAP register with

Format (3):



Description:

The SWAP and SWAPA instructions exchange r[*rd*] with the contents of the word at the addressed memory location.

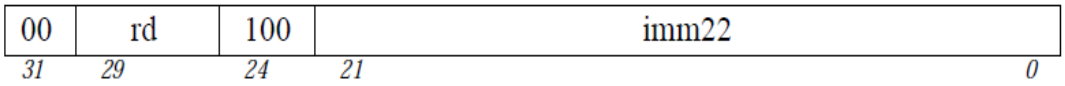
The effective address of a SWAP instruction is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or

"r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one. SWAPA must contain zero in the *i* field, or an illegal_instruction trap will occur. The address space identifier used for the memory accesses is taken from the *asi* field. For SWAP, the address space is either a user or a system data space, according to the S bit in the PSR.

SETHI Instruction

opcode	op	op2	operation
SETHI	00	100	Set High-Order 22 bits

Format (2):



Description:

SETHI zeroes the least significant 10 bits of "r[*rd*]", and replaces its highorder 22 bits with the value from its *imm22* field.

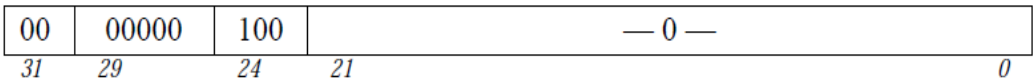
SETHI does not affect the condition codes.

A SETHI instruction with *rd* = 0 and *imm22* = 0 is defined to be a NOP instruction.

NOP Instruction

opcode	op	op2	operation
NOP	00	100	No Operation

Format (2):



Description:

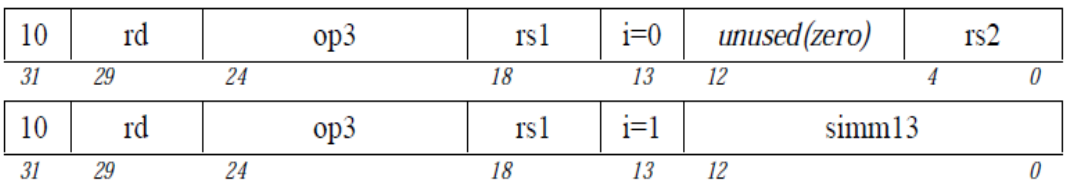
The NOP instruction changes no program-visible state (except the PC and nPC).

Note that NOP is a special case of the SETHI instruction, with *imm22* = 0 and *rd* = 0.

Logical Instructions

opcode	op3	operation
AND	000001	And
ANDcc	010001	And and modify icc
ANDN	000101	And Not
ANDNcc	010101	And Not and modify icc
OR	000010	Inclusive Or
ORcc	010010	Inclusive Or and modify icc
ORN	000110	Inclusive Or Not
ORNcc	010110	Inclusive Or Not and modify icc
XOR	000011	Exclusive Or
XORcc	010011	Exclusive Or and modify icc
XNOR	000111	Exclusive Nor
XNORcc	010111	Exclusive Nor and modify icc

Format (3):



Description:

These instructions implement the bitwise logical operations. They compute "r[*rs1*] **operation** r[*rs2*]" if the *i* field is zero, or "r[*rs1*] **operation** sign_ext(*simm13*)" if the *i* field is one, and write the result into r[*rd*].

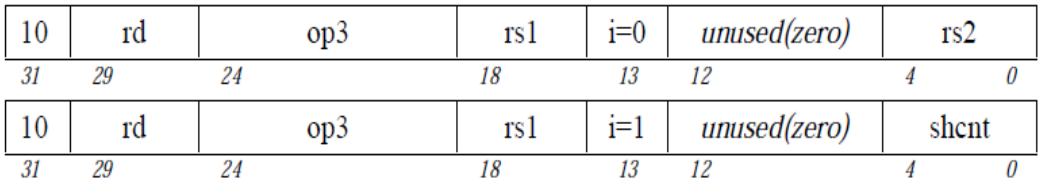
ANDcc, ANDNcc, ORcc, ORNcc, XORcc, and XNORcc modify the integer condition codes (*icc*).

ANDN, ANDNcc, ORN, and ORNcc logically negate their second operand before applying the main (AND or OR) operation.

Shift Instructions

opcode	op3	operation
SLL	100101	Shift Left Logical
SRL	100110	Shift Right Logical
SRA	100111	Shift Right Arithmetic

Format (3):



Description:

The shift count for these instructions is the least significant five bits of r[*rs2*] if the *i* field is zero, or the value in *shcnt* if the *i* field is one.

When *i* is 0, the most significant 27 bits of the value in r[*rs2*] are ignored.When *i* is 1, bits 5 through 12 of the shift instruction are reserved and should be supplied as zero by software.

SLL shifts the value of r[*rs1*] left by the number of bits given by the shift count.

SRL and SRA shift the value of r[*rs1*] right by the number of bits implied by the shift count.

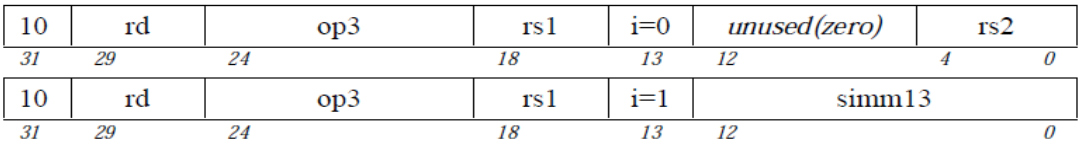
SLL and SRL replace vacated positions with zeroes, whereas SRA fills vacated positions with the most significant bit of r[*rs1*]. No shift occurs when the shift count is zero.

All of these instructions write the shifted result into r[*rd*].These instructions do **not** modify the condition codes.

Add Instructions

opcode	op3	operation
ADD	000000	Add
ADDcc	010000	Add and modify icc

Format (3):



Description:

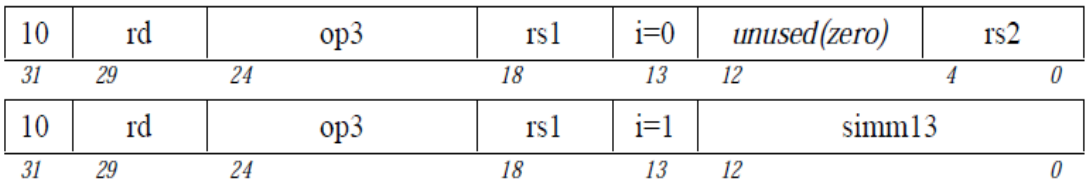
ADD and ADDcc compute "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one, and write the sum into r[*rd*].

ADDcc modifies the integer condition codes (*icc*). Overflow occurs on addition if both operands have the same sign and the sign of the sum is different.

Subtract Instructions

opcode	op3	operation
SUB	000100	Subtract
SUBcc	010100	Subtract and modify icc

Format (3):



Description:

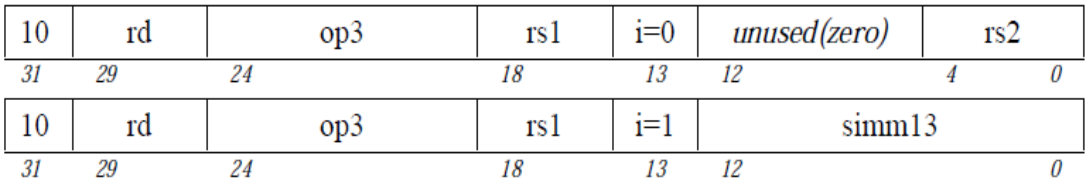
These instructions compute "r[*rs1*] □□r[*rs2*]" if the *i* field is zero,or "r[*rs1*] □□sign_ext(*simm13*)" if the *i* field is one, and write the difference into r[*rd*].

SUBcc modifies the integer condition codes (*icc*). Overflow occurs on subtraction if the operands have different signs and the sign of the difference differs from the sign of r[*rs1*].

SAVE and RESTORE Instructions

opcode	op3	operation
SAVE	111100	Save caller's window
RESTORE	111101	Restore caller's window

Format (3):



Description:

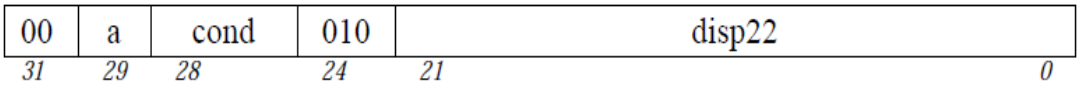
The SAVE instruction subtracts one from the CWP (modulo NWINDOWS) and compares this value (new_CWP) against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is 1, that is, (WIM and 2^{new_CWP}) = 1, then a window_overflow trap is generated. If the WIM bit corresponding to the new_CWP is 0, then no window_overflow trap is generated and new_CWP is written into CWP. This causes the current window to become the CWP−1 window, thereby saving the caller's window.

The RESTORE instruction adds one to the CWP (modulo NWINDOWS) and compares this value (new_CWP) against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is 1, that is, (WIM and 2^{new_CWP}) = 1, then a window_underflow trap is generated. If the WIM bit corresponding to the new_CWP = 0, then no window_underflow trap is generated and new_CWP is written into CWP. This causes the CWP−1 window to become the current window, thereby restoring the caller's window.

Branch on Integer Condition Codes Instructions

opcode	cond	operation	icc test
BNE	1001	Branch on Not Equal	not Z
BE	0001	Branch on Equal	Z
BG	1010	Branch on Greater	not (Z or (N xor V))
C BCS	0101	Branch on Carry Set	

Format (2):



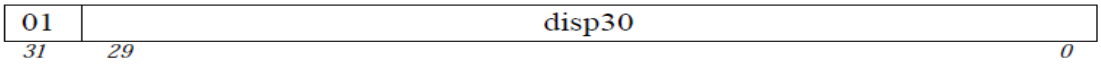
Description:

Conditional Bicc instructions evaluate the integer condition codes (*icc*), according to the *cond* field of the instruction.If "true", the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 □□sign_ext(*disp22*))". If "false", the branch is not taken.

Call and Link Instruction

opcode	op	operation
CALL	01	Call and Link

Format (1):



Description:

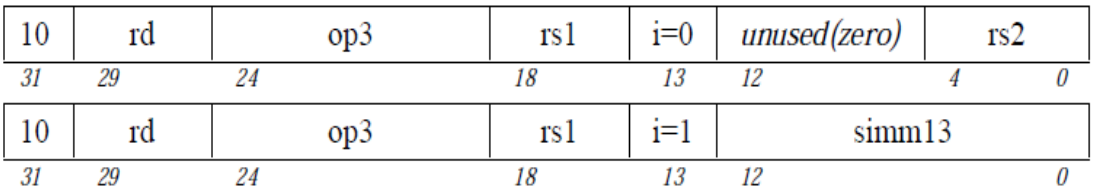
The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address "PC + (4 □□*disp30*)". Since the word displacement (*disp30*) field is 30 bits wide, the target address can be arbitrarily distant. The PC-relative displacement is formed by appending two low-order zeros to the instruction's 30-bit word displacement field.

The CALL instruction also writes the value of PC, which contains the address of the CALL, into r[15] (*out* register 7).

Jump and Link Instruction

opcode	op3	operation
JMPL	111000	Jump and Link

Format (3):



Description:

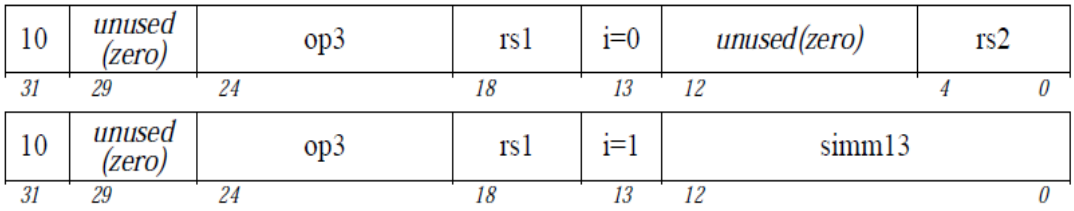
The JMPL instruction causes a register-indirect delayed control transfer to the address given by "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one.

The JMPL instruction copies the PC, which contains the address of the JMPL instruction, into register r[*rd*].

Return from Trap Instruction

opcode	op3	operation
RETT	111001	Return from Trap

Format (3):



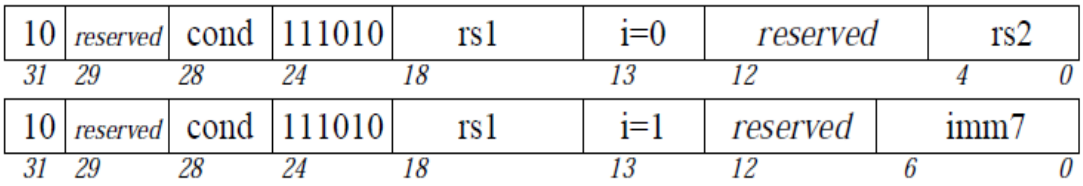
Description:

RETT is used to return from a trap handler. Under some circumstances, RETT may itself cause a trap. If a RETT instruction does not cause a trap, it (1) adds 1 to the CWP (modulo NWINDOWS), (2) causes a delayed control transfer to the target address, (3) restores the S field of the PSR from the PS field, and (4) sets the ET field of the PSR to 1. The target address is "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(*simm13*)" if the *i* field is one.

Trap on Integer Condition Codes Instruction

opcode	cond	operation	cc test
TNE	1001	Trap on Not Equal	**not** Z
TE	0001	Trap on Equal	Z
TG	1010	Trap on Greater	**not** (Z **or** (N **xor** V))
TCS	0101	Trap on Carry Set	C

Format (3):



Description:

A Ticc instruction evaluates the integer condition codes (*icc*) according to the *cond* field of the instruction, producing either a "true" or "false" result.If "true" and no higher priority exceptions or interrupt requests are pending, then a trap_instruction trap is generated. If "false", a trap_instruction trap does not occur and the instruction behaves like a NOP.

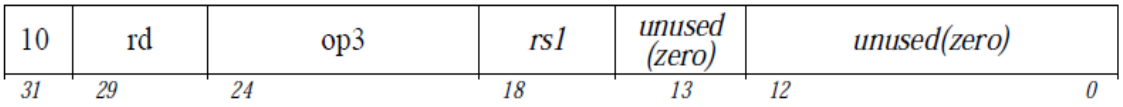
If a trap_instruction trap is generated, the *tt* field of the Trap Base Register (TBR) is written with 128 plus the least significant seven bits of "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or 128 plus the least significant seven bits of "r[*rs1*] + sign_ext" if the *i* field is one.

After a taken Ticc, the processor enters supervisor mode, disables traps,decrements the CWP (modulo NWINDOWS), and saves PC and nPC into r[17] and r[18] (*local* registers 1 and 2) of the new window.

Read State Register Instructions

opcode	op3	rs1	operation
RDPSR Register	101001	0	Read Processor State
RDWIM Mask Register	101010	1	Read Window Invalid
RDTBR	101011	2	Read Trap Base Register
RDASR Register	101000	3	Read Ancillary State

Format (3):



Description:

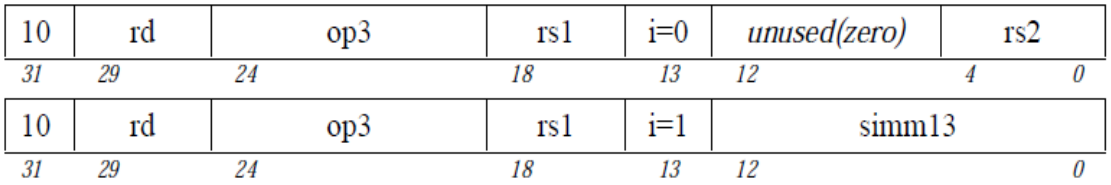
These instructions read the specified IU state register into r[*rd*].

Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trapcontrol registers.

Write State Register Instructions

opcode	op3	rs1	operation
WRPSR Register	110001	0	Write Processor State
WRWIM Invalid Mask Register	110010	1	Write Window
WRTBR Register	110011	2	Write Trap Base
WRASR Register	110000	3	Write Ancillary State

Format (3):



Description:

WRPSR, WRWIM, WRASR, and WRTBR write "r[*rs1*] **xor** r[*rs2*]" if the *I* field is zero, or "r[*rs1*] **xor** sign_ext(*simm13*)" if the *i* field is one, to thewritable fields of the specified IU state register.

Ancillary state registers may include (for example) timer, counter, diagnostic, self-test, and trapcontrol registers.

CHAPTER TWO

MICROARCHITECTURE

2.1 INTRODUCTION

In this chapter, you will learn how to piece together a MIPS micro-processor. Indeed, you will puzzle out three different versions, each with different trade-offs between performance, cost, and complexity.

To the uninitiated, building a microprocessor may seem like black magic. But it is actually relatively straightforward, and by this point you have learned everything you need to know. Specifically, you have learned to design combinational and sequential logic given functional and timing specifications. You are familiar with circuits for arithmetic and memory.

And you have learned about the MIPS architecture, which specifies the programmer's view of the MIPS processor in terms of registers, instruc-tions, and memory.

This chapter covers microarchitecture, which is the connection between logic and architecture. Microarchitecture is the specific arrangement of registers, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture. A particular architecture, such as MIPS, may have many different microarchitectures, each with different trade-offs of performance, cost, and complexity. They all run the same programs, but their internal designs vary widely. We will design three different microarchitectures in this chapter to illustrate the trade-offs.

This chapter draws heavily on David Patterson and John Hennessy's classic MIPS designs in their text Computer Organization and Design. They have generously shared their elegant designs, which have the virtue of illustrating a real commercial architecture while being relatively simple and easy to understand.

2.1.1 Architectural State and Instruction Set

Recall that a computer architecture is defined by its instruction set and architectural state. The architectural state for the MIPS processor consists of the program counter and the 32 registers. Any MIPS microarchitecture must contain all of this state. Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new

architectural state. Some microarchitectures contain additional nonarchitectural state to either simplify the logic or improve performance; we will point this out as it arises.
To keep the microarchitectures easy to understand, we consider only a subset of the MIPS instruction set. Specifically,

We handle the following instructions:
R-type arithmetic/logic instructions: add, sub, and, or, slt
Memory instructions: lw, sw
Branches: beq

After building the microarchitectures with these instructions, we extend them to handle addi and j. These particular instructions were chosen because they are sufficient to write many interesting programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

2.1.2 Design Process

We will divide our microarchitectures into two interacting parts: the datapath and the control. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers.
MIPS is a 32-bit architecture, so we will use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.
A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure 7.1 shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.
In Figure 7.1, heavy lines are used to indicate 32-bit data busses.

Medium lines are used to indicate narrower busses, such as the 5-bit address busses on the register file. Narrow blue lines are used to indicatecontrol signals, such as the register file write enable. We will use this convention throughout the chapter to avoidcluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

The program counter is an ordinary 32-bit register. Its output, PC, points to the current instruction. Its input, PC_, indicates the address of the next instruction.

The instruction memory has a single read port. 1 It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.

The 32-element _ 32-bit register file has two read ports and one write port. The read ports take 5-bit address inputs, A1 and A2, each specifying one of 2 5 _ 32 registers as source operands. They read the 32-bit register values onto read data outputs RD1 and RD2, respectively. The write port takes a 5-bit address input, A3; a 32-bit write data input, WD; a write enable input, WE3; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The data memory has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

The instruction memory, register file, and data memory are all read combinationally. In other words, if the address changes, the new data appears at RD after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must setup sometime before the clock edge and must remain stable until a hold time after the clock edge.

Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. The microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

2.1.3 MIPS Microarchitectures

In this chapter, we develop three microarchitectures for the MIPS processor architecture: single-cycle, multicycle, and pipelined. They differ in the way that the state elements are connected together and in the amount of nonarchitectural state.

The single-cycle microarchitecture executes an entire instruction in one cycle. It is easy to explain and has a simple control unit.

Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction.

The multicycle microarchitecture executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories. For example, the adder may be used on several different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multi-
ple clock cycles.

The pipelined microarchitecture applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

We explore the details and trade-offs of these three microarchitectures in the subsequent sections. At the end of the chapter, we briefly mention additional techniques that are used to get even more speed in modern high-performance microprocessors.

2.2 PERFORMANCE ANALYSIS

As we mentioned, a particular processor architecture can have many microarchitectures with different cost and performance trade-offs. The cost depends on the amount of hardware required and the implementation

technology. Each year, CMOS processes can pack more transistors on a chip for the same amount of money, and processors take advantage of these additional transistors to deliver more performance. Precise cost calculations require detailed knowledge of the implementation technology, but in general, more gates and more memory mean more dollars.

This section lays the foundation for analyzing performance.

There are many ways to measure the performance of a computer system, and marketing departments are infamous for choosing the method that makes their computer look fastest, regardless of whether the measurement has any correlation to real world performance. For example, Intel and Advanced Micro Devices (AMD) both sell compatible microprocessors conforming to the IA-32 architecture. Intel Pentium III and Pentium 4 microprocessors were largely advertised according to clock frequency in the late 1990s and early 2000s, because Intel offered higher clock frequencies than its competitors. However, Intel's main competitor, AMD, sold Athlon microprocessors that executed programs faster than Intel's chips at the same clock frequency. What is a consumer to do?

The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run; this may be necessary if you haven't written your program yet or if somebody else who doesn't have your program is making the measurements. Such collections of programs are called benchmarks, and the execution times of these programs are commonly published to give some indication of how a processor performs.

The execution time of a program, measured in seconds, is given by Equation 7.1.

$$Execution\ Time = \left(\#\ instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \quad (7.1)$$

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware. The number of instructions also depends enormously on the cleverness of the

programmer. For the purposes of this chapter, we will assume that we are executing known programs on a MIPS processor, so the number of instructions for each program is constant, independent of the microarchitecture.

The number of cycles per instruction, often called CPI, is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (instructions per cycle, or IPC). Different microarchitectures have different CPIs. In this chapter, we will assume we have an ideal memory system that does not affect the CPI. In Chapter 8, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

The number of seconds per cycle is the clock period, Tc. The clock period is determined by the critical path through the logic on the processor. Different microarchitectures have different clock periods. Logic and circuit designs also significantly affect the clock period. For example, a carry-lookahead adder is faster than a ripple-carry adder. Manufacturing advances have historically doubled transistor speeds every 4-6 years, so a microprocessor built today will be much faster than one from last decade, even if the microarchitecture and logic are unchanged.

The challenge of the microarchitect is to choose the design that minimizes the execution time while satisfying constraints on cost and/or power consumption. Because microarchitectural decisions affect both CPI and T c and are influenced by logic and circuit designs, determining the best choice requires careful analysis.

There are many other factors that affect overall computer performance. For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors that make processor performance irrelevant. The fastest microprocessor in the world doesn't help surfing the Internet on a dial-up connection. But these other factors are beyond the scope of this book.

2.3 SINGLE-CYCLE PROCESSOR

We first design a MIPS microarchitecture that executes instructions in a single cycle. We begin constructing the datapath by connecting the state elements from Figure 7.1 with combinational logic that can execute the various instructions. Control signals determine which specific instruction is carried out by the datapath at any given time. The controller contains combinational logic

that generates the appropriate control signals based on the current instruction. We conclude by analyzing the performance of the single-cycle processor.

2.3.1 Single-Cycle Datapath

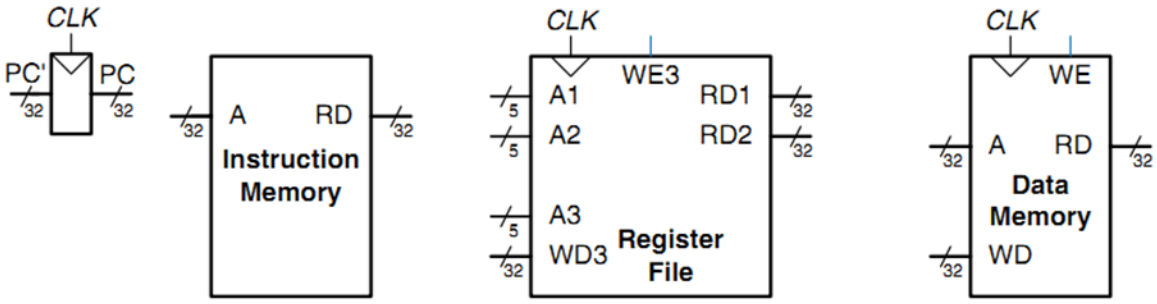


Figure 7.1 State elements of MIPS processor

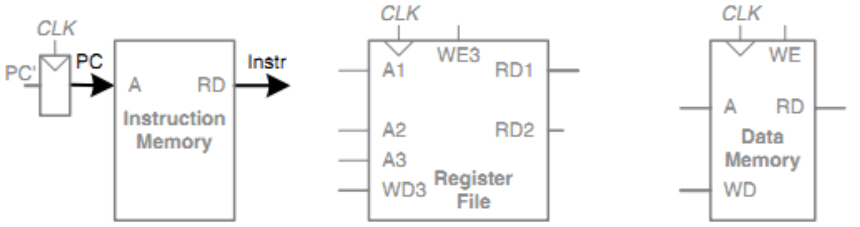


Figure 7.2 Fetch instruction from memory

This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements from Figure 7.1. The new connections are emphasized in black (or blue, for new control signals), while the hardware that has already been studied is shown in gray.

The program counter (PC) register contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.2 shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr.

The processor's actions depend on the specific instruction that was fetched. First we will work out the datapath connections for the lw instruction. Then we will consider how to generalize the datapath to handle the other instructions.

For a lw instruction, the next step is to read the source register containing the base address. This register is specified in the rs field of the instruction, Instr25:21 . These bits of the instruction are connected to the address input of one of the register file read ports, A1, as shown in Figure 7.3. The register file reads the register value onto RD1.

The lw instruction also requires an offset. The offset is stored in the immediate field of the instruction, Instr15:0 . Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits, as shown in Figure 7.4. The 32-bit sign-extended value is called SignImm. Recall from Section 1.4.6 that sign extension simply copies the sign bit (most significant bit) of a short input into all of the upper bits of the longer output. Specifically, SignImm15:0 _ Instr15:0 and SignImm31:16 _ Instr15.

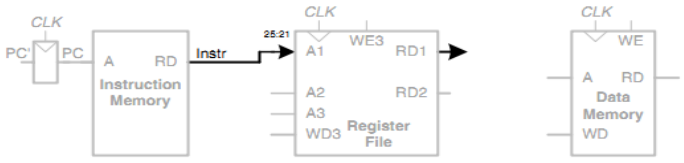


Figure 7.3 Read source operand from register file

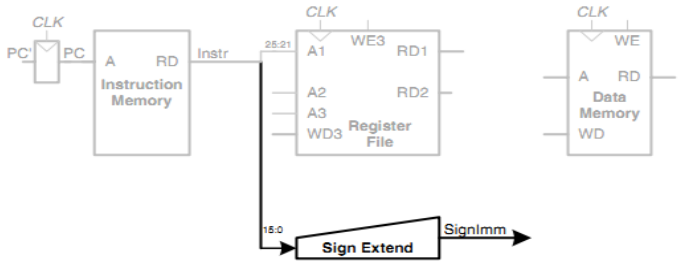


Figure 7.4 Sign-extend the immediate

The processor must add the base address to the offset to find the address to read from memory. Figure 7.5 introduces an ALU to perform this addition. The ALU receives two operands, SrcA and SrcB. SrcA comes from the register file, and SrcB comes from the sign-extended immediate. The ALU can perform many operations, as was described in Section 5.2.4. The 3-bit ALUControl signal specifies the operation. The ALU generates a 32-bit ALUResult and a Zero flag, that indicates whether ALUResult __ 0. For a lw to add the base instruction, the ALUControl signal should be set to 010 address and offset. ALUResult is sent to the data memory as the address for the load instruction, as shown in Figure 7.5.

The data is read from the data memory onto the ReadData bus, then written back to the destination register in the register file at the end of the cycle, as shown in Figure 7.6. Port 3 of the register file is the write port.

The destination register for the lw instruction is specified in the rt field, Instr20:16 , which is connected to the port 3 address input, A3, of the register file.

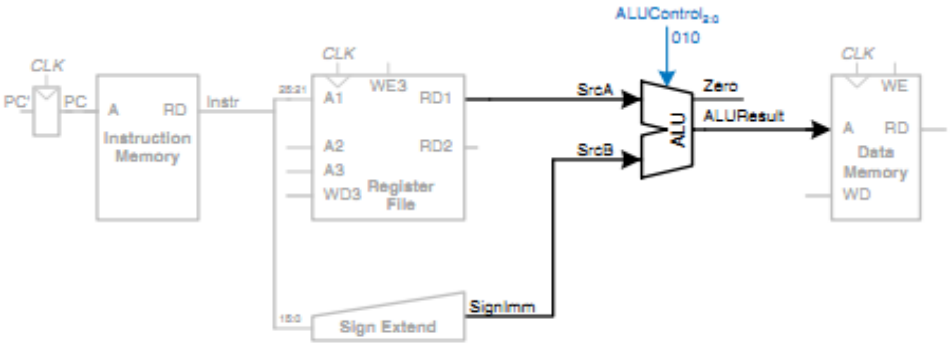


Figure 7.5 Compute memory address

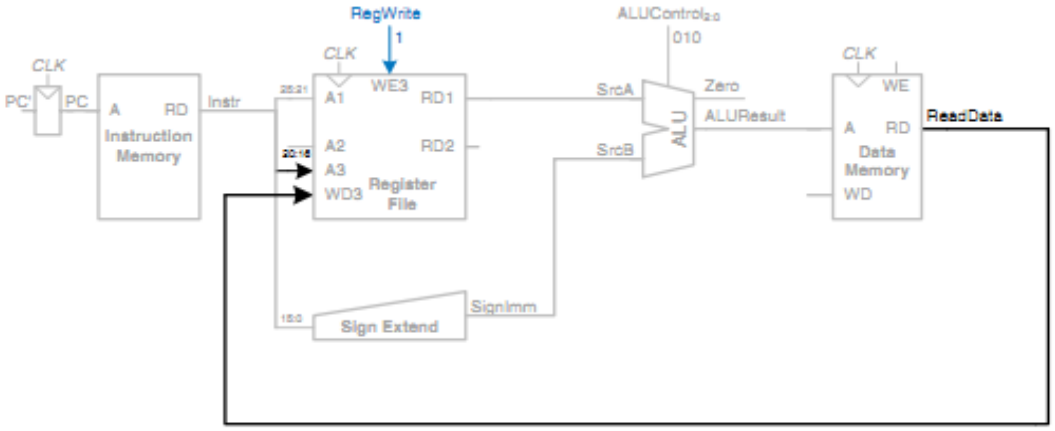


Figure 7.6 Write data back to register file

The ReadData bus is connected to the port 3 write data input, WD3, of the register file. A control signal called RegWrite is connected to the port 3 write enable input, WE3, and is asserted during a lw instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle.

While the instruction is being executed, the processor must compute the address of the next instruction, PC_. Because instructions are 32 bits _ 4 bytes, the next instruction is at PC _ 4. Figure 7.7 uses another adder to increment the PC by 4. The new address is written into the program counter on the next rising edge of the clock.

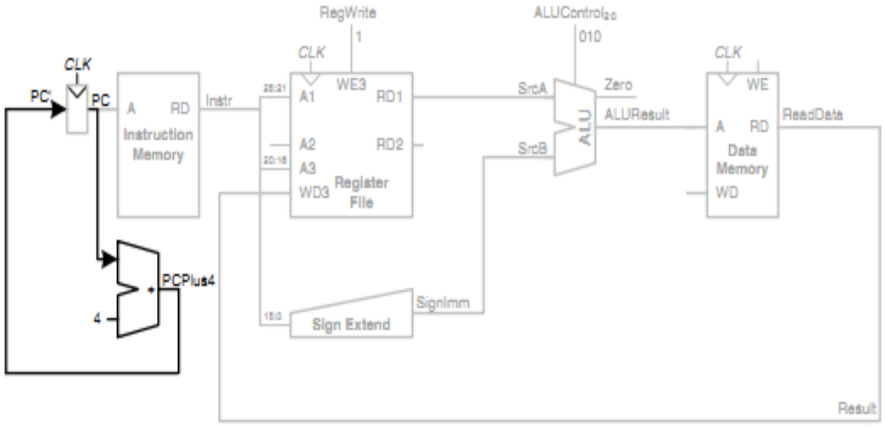


Figure 7.7 Determine address of next instruction for PC

This completes the datapath for the lw instruction. Next, let us extend the datapath to also handle the sw instruction. Like the lw instruction, the sw instruction reads a base address from port 1 of the register and sign-extends an immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by the datapath.

The sw instruction also reads a second register from the register file and writes it to the data memory. Figure 7.8 shows the new connections forthis function. The register is specified in the rt field, Instr20:16 of the instruction are connected to the second register file read port, A2.

The register value is read onto the RD2 port. It is connected to the write data port of the data memory.

The write enable port of the data memory, WE, is controlled by MemWrite. For a sw instruction, MemWrite _ 1, to write the data to memory; ALUControl _ 010, to add the base address and offset; and RegWrite _ 0, because nothing should be written to the register file.

Note that data is still read from the address given to the data memory, but that this ReadData is ignored because RegWrite _ 0.
Next, consider extending the datapath to handle the R-type instructions add, sub, and, or, and slt. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different ALUControl signals. Figure 7.9 shows the enhanced datapath handling R-type instructions. The register file reads two registers. The ALU performs an operation on these two registers. In Figure 7.8, the ALU always received its SrcB operand from the sign-extended immediate (SignImm). Now, we add a multiplexer to choose SrcB from either the register file RD2 port or SignImm.

The multiplexer is controlled by a new signal, ALUSrc. ALUSrc is 0 for R-type instructions to choose SrcB from the register file; it is 1 for lw and sw to choose SignImm. This principle of enhancing the datapath's capabilities by adding a multiplexer to choose inputs from several possibilities is extremely useful. Indeed, we will apply it twice more to complete the handling of R-type instructions.

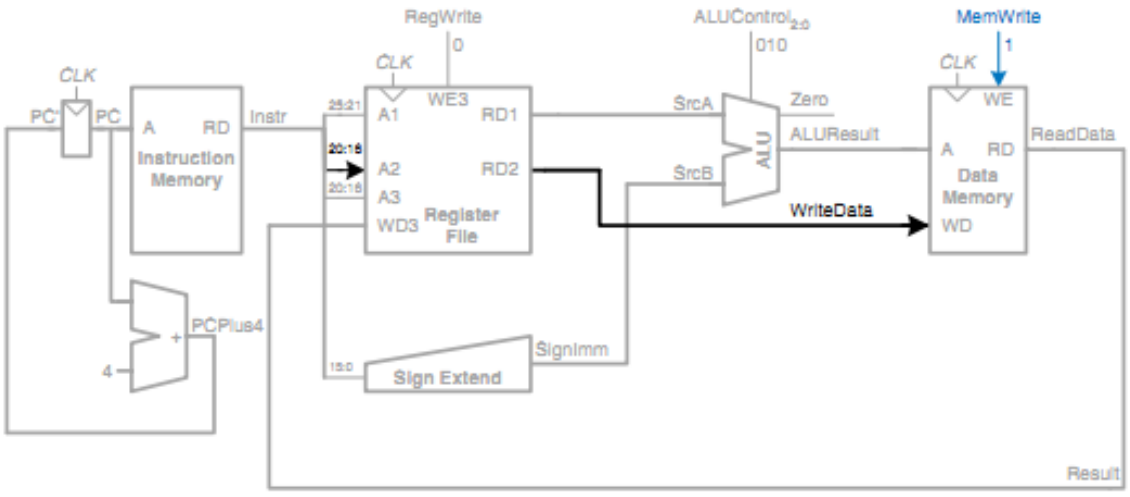


Figure 7.8 Write data to memory for sw instruction

In Figure 7.8, the register file always got its write data from the data memory. However, R-type instructions write the ALUResult to the register file. Therefore, we add another multiplexer to choose between ReadData and ALUResult. We call its output Result. This multiplexer is controlled by another new signal, MemtoReg. MemtoReg is 0 for R-type instructions to choose Result from the ALUResult; it is 1 for lw to choose ReadData. We don't care about the value of MemtoReg for sw, because sw does not write to the register file. Similarly, in Figure 7.8, the register to write was specified by the rt field of the instruction, Instr20:16 . . However, for R-type instructions, the register is specified by the rd field, Instr15:11 . . Thus, we add a third multiplexer to choose WriteReg from the appropriate field of the instruction.

The multiplexer is controlled by RegDst. RegDst is 1 for R-type instructions to choose WriteReg from the rd field, Instr15:11 ; it is 0 for lw to choose the rt field, Instr20:16 . We don't care about the value of RegDst for sw, because sw does not write to the register file.
Finally, let us extend the datapath to handle beq. beq compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter.

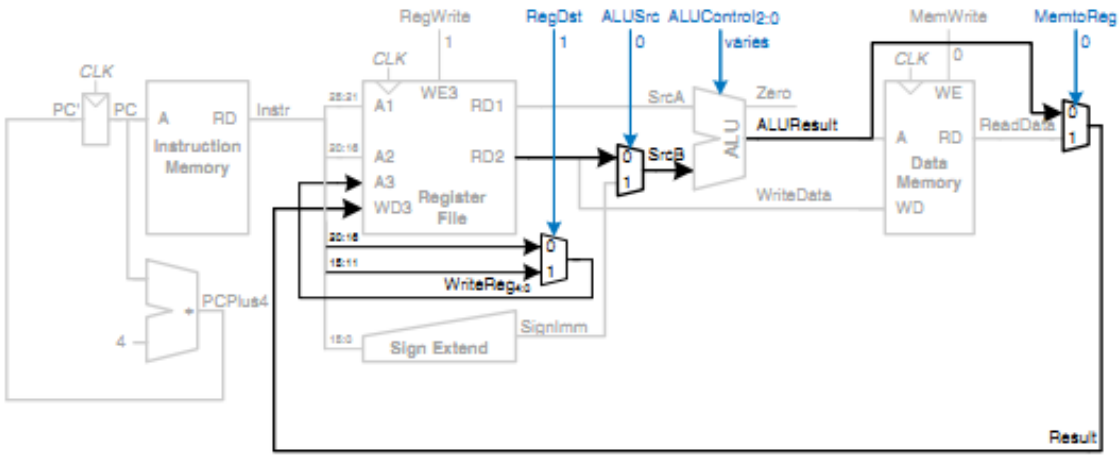


Figure 7.9 Datapath enhancements for R-type instruction

Recall that the offset is a positive or negative number, stored in the imm field of the instruction, Instr31:26 .The offset indicates the number of instructions to branch past. Hence, the immediate must be sign-extended and multiplied by 4 to get the new program counter value:

PC_ _ PC _ 4 _ SignImm _ 4.Figure 7.10 shows the datapath modifications. The next PC value for a taken branch, PCBranch, is computed by shifting SignImm left by 2 bits, then adding it to PCPlus4. The left shift by 2 is an easy way to multiply by 4, because a shift by a constant amount involves just wires.

The two registers are compared by computing SrcA _ SrcB using the ALU. If ALUResult is 0, as indicated by the Zero flag from the ALU, the registers are equal.

We add a multiplexer to choose PC_ from either PCPlus4 or PCBranch. PCBranch is selected if the instruction is a branch and the Zero flag is asserted. Hence, Branch is 1 for beq and 0 for other instructions.

For beq, ALUControl _ 110, so the ALU performs a subtraction. ALUSrc _ 0 to choose SrcB from the register file. RegWrite and MemWrite are 0, because a branch does not write to the register file or memory. We don't care about the values of RegDst and MemtoReg, because the register file is not written.

This completes the design of the single-cycle MIPS processor datapath. We have illustrated not only the design itself, but also the design process in which the state elements are identified and the combinational logic connecting

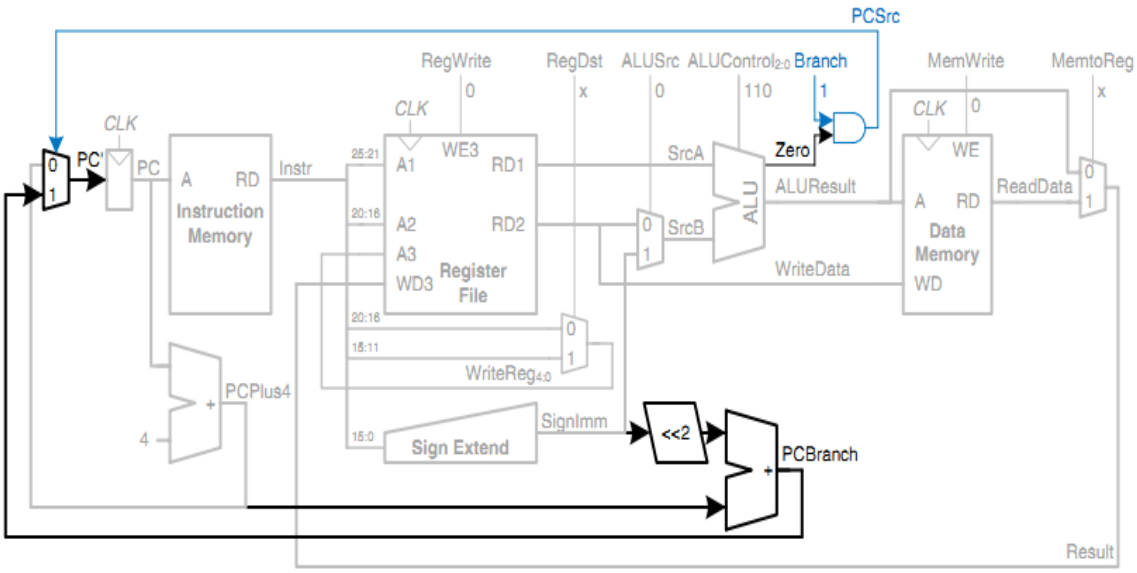


Figure 7.10 Datapath enhancements for beq instruction

the state elements is systematically added. In the next section, we consider how to compute the control signals that direct the operation of our datapath.

2.3.2 Single-Cycle Control

The control unit computes the control signals based on the opcode and funct fields of the instruction, Instr31:26 and Instr5:0. Figure 7.11 shows the entire single-cycle MIPS processor with the control unit attached to the datapath.

Most of the control information comes from the opcode, but R-type instructions also use the funct field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure 7.12The main decoder computes most of the outputs from the opcode. It also determines a 2-bit ALUOp signal.

The ALU decoder uses this ALUOp signal in conjunction with the funct field to compute ALUControl. The meaning of the ALUOp signal is given in Table 7.1.

Table 7.1 *ALUOp* encoding

ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

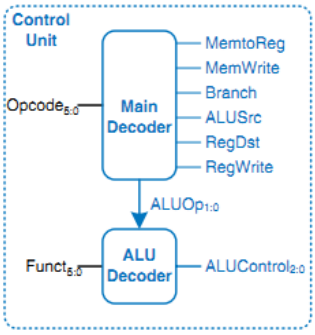


Figure 7.12 Control unit internal structure

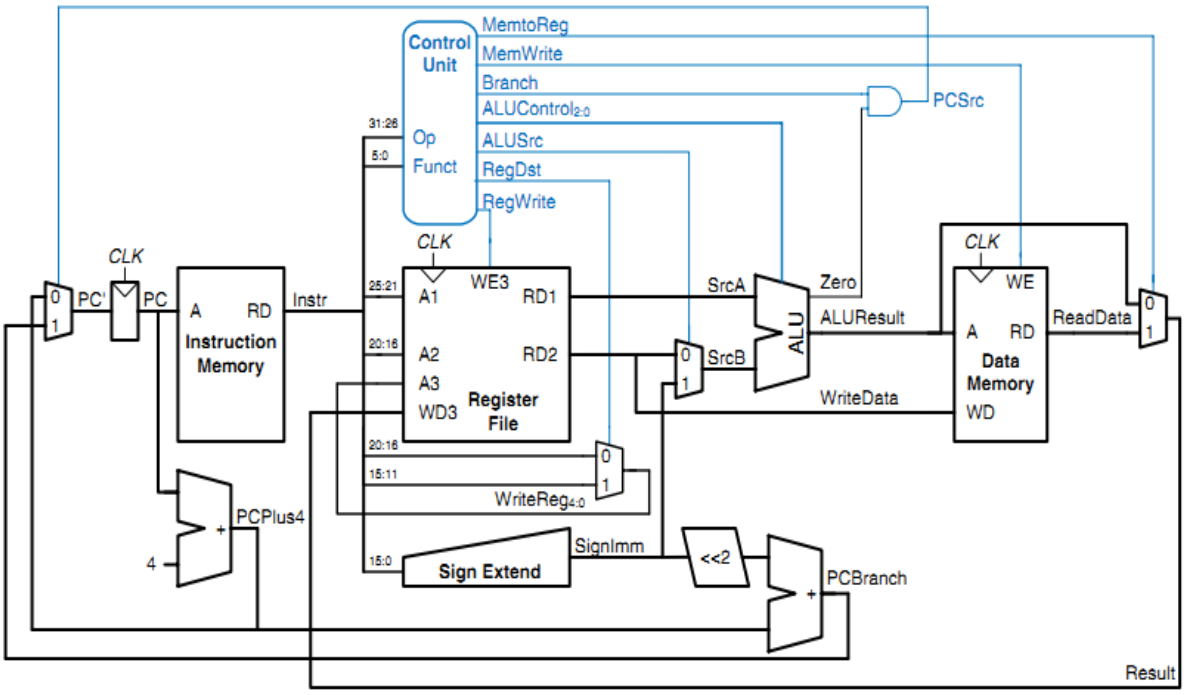


Figure 7.11 Complete single-cycle MIPS processor

Table 7.2 ALU decoder truth table

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

Table 7.3 Main decoder truth table

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Table 7.2 is a truth table for the ALU decoder. Recall that the meanings of the three ALUControl signals were given in Table 5.1. Because ALUOp is never 11, the truth table can use don't care's X1 and 1X instead of 01 and 10 to simplify the logic. When ALUOp is 00 or 01, the ALU should add or subtract, respectively. When ALUOp is 10, the decoder examines the funct field to determine the ALUControl. Note that, for the R-type instructions we implement, the first two bits of the funct field are always 10, so we may ignore them to simplify the decoder.

The control signals for each instruction were described as we built the datapath. Table 7.3 is a truth table for the main decoder that summarizes the control signals as a function of the opcode. All R-type instructions use the same main decoder values; they differ only in the ALU decoder output. Recall that, for instructions that do not write to the register file (e.g., sw and beq), the RegDst and MemtoReg control signals are don't cares (X); the address and data to the register write port do not matter because RegWrite is not asserted. The logic for the decoder can be designed using your favorite techniques for combinational logic design.

Example SINGLE-CYCLE PROCESSOR OPERATION

Determine the values of the control signals and the portions of the datapath that are used when executing an or instruction.

Solution: Figure 7.13 illustrates the control signals and flow of data during execution of the or instruction. The PC points to the memory location holding the instruction, and the instruction memory fetches this instruction.

The main flow of data through the register file and ALU is represented with a dashed blue line. The register file reads the two source operands specified by Instr25:21 and Instr20:16 . SrcB should come from the second port of the register file (not SignImm), so ALUSrc must be 0. or is an R-type instruction, so ALUOp is 10, indicating that ALUControl should be determined from the funct field to be 001. Result is taken from the ALU, so MemtoReg is 0. The result is written to the register file, so RegWrite is 1. The instruction does not write memory, so MemWrite _ 0.The selection of the destination register is also shown with a dashed blue line. The destination register is specified in the rd field, Instr15:11 , so RegDst 1.

The updating of the PC is shown with the dashed gray line. The instruction is not a branch, so Branch _ 0 and, hence, PCSrc is also 0. The PC gets its next value from PCPlus4.

Note that data certainly does flow through the nonhighlighted paths, but that

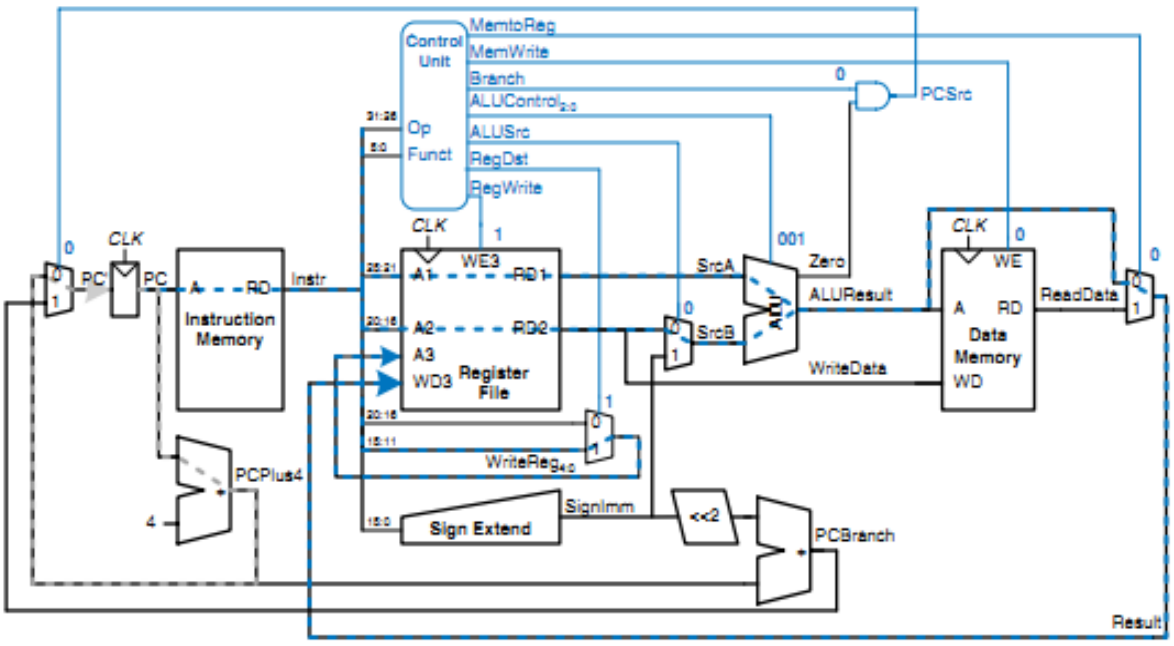


Figure 7.13 Control signals and data flow while executing or instruction

the value of that data is unimportant for this instruction. For example, the immediate is sign-extended and data is read from memory, but these values do not influence the next state of the system.

2.3.3 More Instructions

We have considered a limited subset of the full MIPS instruction set. Adding support for the addi and j instructions illustrates the principle of how to handle new instructions and also gives us a sufficiently rich instruction set to write many interesting programs. We will see that supporting some instructions simply requires enhancing the main decoder, whereas supporting others also requires more hardware in the datapath.

Example addi INSTRUCTION

The add immediate instruction, addi, adds the value in a register to the immediate and writes the result to another register. The datapath already is capable of this task. Determine the necessary changes to the controller to support addi.

Solution: All we need to do is add a new row to the main decoder truth table showing the control signal values for addi, as given in Table 7.4. The result should be written to the register file, so RegWrite _ 1. The destination register is specified in the rt field of the instruction, so RegDst _ 0. SrcB comes from the immediate, so ALUSrc _ 1. The instruction is not a branch, nor does it write memory, so Branch _ MemWrite _ 0. The result comes from the ALU, not memory, so MemtoReg _ 0. Finally, the ALU should add, so ALUOp _ 00.

Table 7.4 Main decoder truth table enhanced to support addi

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

Example j INSTRUCTION

The jump instruction, j, writes a new value into the PC. The two least significant bits of the PC are always 0, because the PC is word aligned (i.e., always a multiple of 4). The next 26 bits are taken from the jump address field in Instr25:0 . The upper four bits are taken from the old value of the PC. The existing datapath lacks hardware to compute PC_ in this fashion. Determine the necessary changes to both the datapath and controller to handle j.

Solution: First, we must add hardware to compute the next PC value, PC_, in the case of a j instruction and a multiplexer to select this next PC, as shown in Figure 7.14. The new multiplexer uses the new Jump control signal.

Now we must add a row to the main decoder truth table for the j instruction and a column for the Jump signal, as shown in Table 7.5. The Jump control signal is 1 for the j instruction and 0 for all others. j does not write the register file or memory, so RegWrite _ MemWrite _ 0. Hence, we don't care about the computation done in the datapath, and RegDst _ ALUSrc _ Branch _ MemtoReg _ ALUOp _ X.

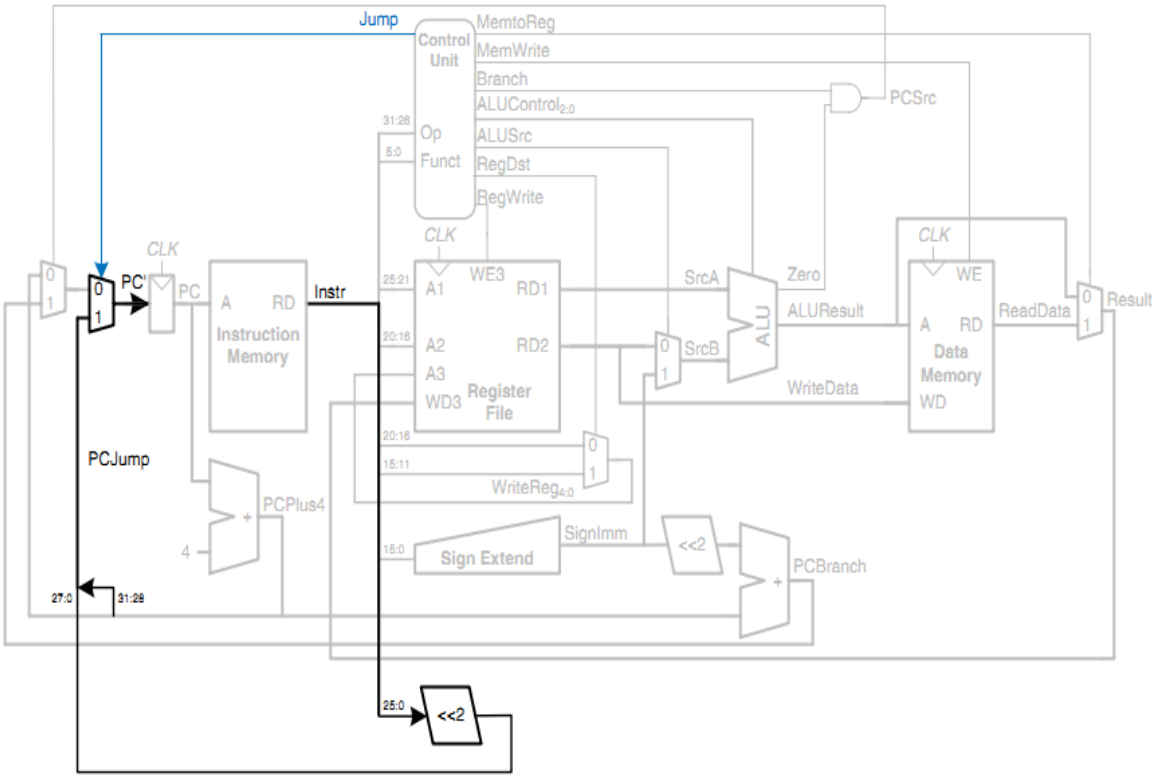


Figure 7.14 Single-cycle MIPS datapath enhanced to support the j instruction

2.3.4 Performance Analysis

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1. The critical path for the lw instruction is shown in Figure 7.15 with a heavy dashed blue line. It starts with the PC loading a new address on the rising edge of the clock. The instruction memory reads the next instruction. The register file reads SrcA. While the register file is reading, the immediate field is sign-extended and selected at the ALUSrc multiplexer to determine SrcB. The ALU adds SrcA and SrcB to find the effective address. The data memory reads from this address. The MemtoReg multiplexer selects ReadData. Finally, Result must setup at the register file before the next rising clock edge, so that it can be properly written. Hence, the cycle time is

$$T_c = t_{pcq_PC} + t_{mem} + \max[t_{RFread}, t_{sext}] + t_{mux} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \quad (7.2)$$

$$T_c = t_{pcq_PC} + 2t_{mem} + t_{RFread} + 2t_{mux} + t_{ALU} + t_{RFsetup} \quad (7.3)$$

Table 7.5 Main decoder truth table enhanced to support j

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

In most implementation technologies, the ALU, memory, and register file accesses are substantially slower than other operations. Therefore, the cycle time simplifies to

The numerical values of these times will depend on the specific implementation technology.

Other instructions have shorter critical paths. For example, R-type instructions do not need to access data memory. However, we are disciplining ourselves to synchronous sequential design, so the clock period is constant and must be long enough to accommodate the slowest instruction.

Example SINGLE-CYCLE PROCESSOR PERFORMANCE
Ben Bitdiddle is contemplating building the single-cycle MIPS processor in a 65 nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.6. Help him compare the execution time for a program with 100 billion instructions.

Solution: According to Equation 7.3, the cycle time of the single-cycle processor is Tc1 _ 30 _ 2(250) _ 150 _ 2(25) _ 200 _ 20 _ 950 ps. We use the subscript "1" to distinguish it from subsequent processor designs. According to Equation 7.1, the total execution time is T 1 _ (100 _ 10 9 instructions)(1 cycle/instruction)

Table 7.6 Delays of circuit elements

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	200
memory read	t_{mem}	250
register file read	t_{RFread}	150
register file setup	$t_{RFsetup}$	20

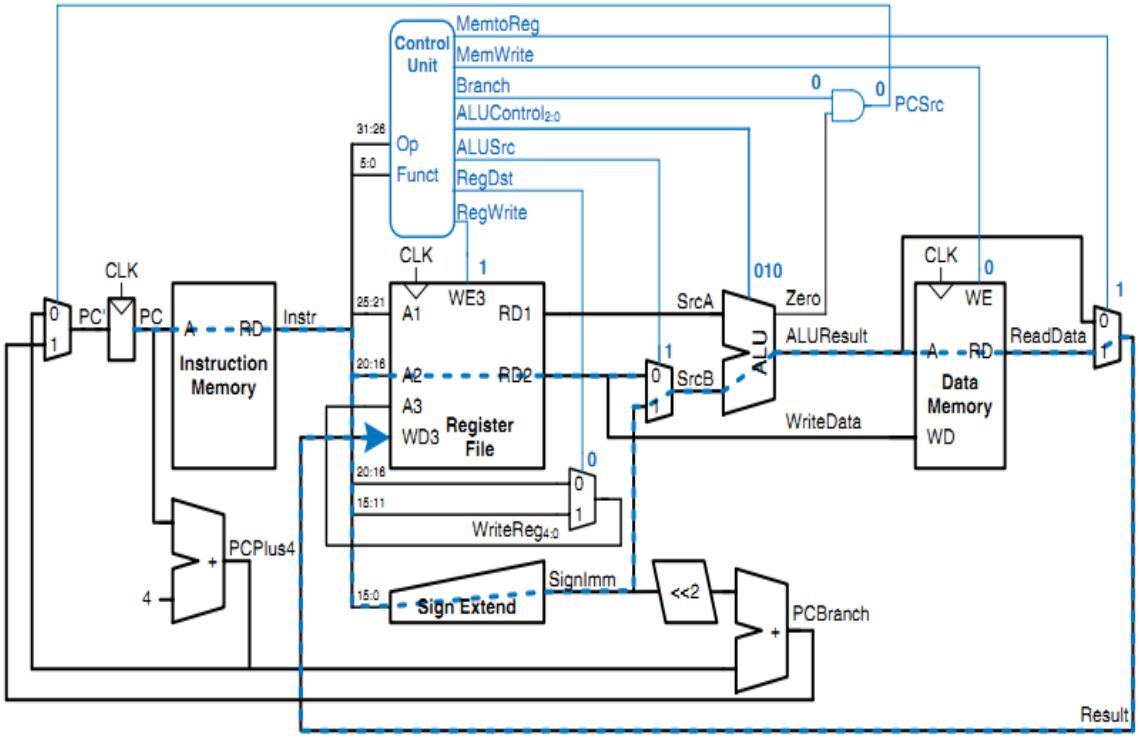


Figure 7.15 Critical path for `lw` instruction

CHAPTER TRHEE

CODING

--Adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity adder is
  port(a: in unsigned(31 downto 0);
       b: in unsigned(31 downto 0);
       c: out unsigned(31 downto 0));
 end adder;

 architecture arc_adder of adder is
 begin

   c<=a + b;

 end arc_adder;
```

--right_shift

```
library ieee;
use ieee.std_logic_1164.all;

entity r_shift is
  port(a : in std_logic_vector(31 downto 0);
       sel : in integer range 0 to 31;
       c : out std_logic_vector(31 downto 0));
end r_shift;

architecture arc_r_shift of r_shift is
subtype vector is std_logic_vector(31 downto 0);
type matrix is array(31 downto 0) of vector;
signal row : matrix;
begin
  row(0)<=a;
  g_1 : for i in 1 to 31 generate
          row(i)<='0' & row(i-1)(31 downto 1);
        end generate;
        c<=row(sel);
end arc_r_shift;
```

--arithmatic_right_shift

```
library ieee;
use ieee.std_logic_1164.all;

entity a_r_shift is
  port(a : in std_logic_vector(31 downto 0);
       sel : in integer range 0 to 31;
```

```
      c : out std_logic_vector(31 downto 0));
end a_r_shift;

architecture arc_a_r_shift of a_r_shift is
subtype vector is std_logic_vector(31 downto 0);
type matrix is array(31 downto 0) of vector;
signal row : matrix;
signal s_1 : std_logic;
begin
  row(0)<=a;
  g_1 : for i in 1 to 31 generate
          row(i)<=a(31) & row(i-1)(31 downto 1);
        end generate;

        c<=row(sel);

end arc_a_r_shift;
```

--arithmatic_left_shift

```
library ieee;
use ieee.std_logic_1164.all;

entity l_shift is
  port(a : in std_logic_vector(31 downto 0);
       sel : in integer range 0 to 31;
       c : out std_logic_vector(31 downto 0));
end l_shift;

architecture arc_l_shift of l_shift is
subtype vector is std_logic_vector(31 downto 0);
type matrix is array(31 downto 0) of vector;
signal row : matrix;
begin
  row(0)<=a;
  g_1 : for i in 1 to 31 generate
          row(i)<=row(i-1)(30 downto 0) & '0';
        end generate;

        c<=row(sel);
end arc_l_shift;
```

--arithmatic_shift

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity arith_sh is
  port(a_1: in unsigned(31 downto 0);
       b_1: in unsigned(31 downto 0);
       sel: in unsigned(2 downto 0);
       c: out unsigned(31 downto 0);
       negative: out std_logic;
       zero: out std_logic;
       over_flow: out std_logic;
```

```
       carry:out std_logic;
       borrow: out std_logic);
end arith_sh;

architecture arc_arith_sh of arith_sh is
signal s_1,s_2,s_3,s_4,a,b : std_logic_vector(31 downto 0);
signal sel_sh: integer range 0 to 31;
begin
  a<=conv_std_logic_vector(a_1,32);
  b<=conv_std_logic_vector(b_1,32);
  sel_sh<=conv_integer(b_1(4 downto 0));

  u1: entity work.l_shift(arc_l_shift) port
map(a=>a,sel=>sel_sh,c=>s_2);
  u2: entity work.r_shift(arc_r_shift) port
map(a=>a,sel=>sel_sh,c=>s_3);
  u3: entity work.a_r_shift(arc_a_r_shift) port
map(a=>a,sel=>sel_sh,c=>s_4);


  with sel select
  s_1<=a+b when "000",
     a+((not b)+1) when "001",
     s_2 when "010",
     s_3 when "011",
     s_4 when "100",
     (others=>'0') when others;

  zero<='1' when s_1=conv_std_logic_vector(0,32) else
        '0';

negative<=s_1(31);

over_flow<=(((not a(31)) and (not b(31)) and s_1(31)) or (a(31) and
b(31) and (not s_1(31))))
or (((not a(31)) and b(31) and s_1(31)) or (a(31) and (not b(31)) and
(not s_1(31)))); -- with overflow enable

carry<=((not a(31)) and (not b(31)) and s_1(31)) or (a(31) and b(31)
and (not s_1(31)));          -- with add enable

borrow<=((not a(31)) and (not b(31)) and s_1(31)) or (a(31) and (not
b(31)) and s_1(31));         -- with sub enable

c<=unsigned(s_1);

end arc_arith_sh;
```

--logic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity logic is
      port(a_1 : in unsigned(31 downto 0);
           b_1 : in unsigned(31 downto 0);
           sel : in unsigned(2 downto 0);
           c : out unsigned(31 downto 0));
```

```
end logic;

architecture arc_logic of logic is
signal a,b,s_1 : std_logic_vector(31 downto 0);
begin

  a<=conv_std_logic_vector(a_1,32);
  b<=conv_std_logic_vector(b_1,32);

with sel select
     s_1<= a and b when "000",
           a or b when "001",
           a xor b when "010",
           a nand b when "011",
           a nor b when "100",
           a xnor b when "101",
           (others=>'0') when others;

          c<=unsigned(s_1);
end arc_logic;
```

--ALU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity alu is
  port(a: unsigned(31 downto 0);
       b: unsigned(31 downto 0);
       sel: in unsigned(3 downto 0);
       c: out unsigned(31 downto 0);
       negative: out std_logic;
       zero: out std_logic;
       over_flow: out std_logic;
       carry:out std_logic;
       borrow: out std_logic);
  end alu;

architecture arc_alu of alu is
signal s_1,s_2:unsigned(31 downto 0);
begin

  u1: entity work.arith_sh(arc_arith_sh) port map(a_1=>a,b_1=>b,sel=>
sel(2 downto
0),c=>s_1,negative=>negative,zero=>zero,over_flow=>over_flow,carry=>c
arry,
borrow=>borrow);

  u2: entity work.logic(arc_logic) port map(a_1=>a,b_1=>b,sel=>sel(2
downto 0),c=>s 2);

  with sel(3) select
   c<=s_1 when '0',
      s_2 when others;

end arc_alu;
```

--instruction memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity rom is
port (instr_addr : in unsigned(7 downto 0);
instr_out : out unsigned(31 downto 0)
);
end rom;

architecture arc rom of rom is
type mem_type is array(0 to 255) of unsigned(7 downto 0);
signal mem : mem_type;
signal s_1,s_2,s_3: unsigned(7 downto 0);
begin
----------------------Program-----------------------------------

mem(3)<=x"82";mem(2)<=x"00";mem(1)<=x"20";mem(0)<=x"05";           --
ADD      simm=5
mem(7)<=x"84";mem(6)<=x"20";mem(5)<=x"20";mem(4)<=x"05";           --
SUB      simm=-5
mem(11)<=x"86";mem(10)<=x"80";mem(9)<=x"40";mem(8)<=x"02";         --
ADDCC
mem(15)<=x"02";mem(14)<=x"80";mem(13)<=x"00";mem(12)<=x"04";       --
BE  to  28
mem(19)<=x"06";mem(18)<=x"00";mem(17)<=x"40";mem(16)<=x"02";       --
add
mem(23)<=x"08";mem(22)<=x"00";mem(21)<=x"40";mem(20)<=x"02";       --
sub
mem(27)<=x"00";mem(26)<=x"00";mem(25)<=x"00";mem(24)<=x"02";       --
branch
mem(31)<=x"92";mem(30)<=x"00";mem(29)<=x"20";mem(28)<=x"f1";       --
ADD      28
mem(35)<=x"81";mem(34)<=x"e0";mem(33)<=x"20";mem(32)<=x"01";       --
SAVE
mem(39)<=x"f2";mem(38)<=x"a0";mem(37)<=x"00";mem(36)<=x"01";       --
STA
mem(43)<=x"40";mem(42)<=x"00";mem(41)<=x"00";mem(40)<=x"09";       --
CALL  36
mem(47)<=x"40";mem(46)<=x"00";mem(45)<=x"00";mem(44)<=x"09";       --
```

```
--------------interrupt routine--------------------------------

mem(51)<=x"cc";mem(50)<= x"80";mem(49)<= x"00";mem(48)<= x"00";    --
LDA   rd=6
mem(55)<=x"b2";mem(54)<= x"26";mem(53)<= x"60";mem(52)<= x"01";    --
SUB   i=1  rd,rs=25   simm=1
mem(59)<=x"cc";mem(58)<= x"a0";mem(57)<= x"00";mem(56)<= x"01";    --
STA
mem(63)<=x"8e";mem(62)<= x"00";mem(61)<= x"20";mem(60)<= x"0f";    --
ADD i=1   rd=7   simm=15
mem(67)<=x"8e";mem(66)<= x"a1";mem(65)<= x"e0";mem(64)<= x"01";    --
SUBCC i=1   rs=7  rd=7
mem(71)<=x"12";mem(70)<= x"bf";mem(69)<= x"ff";mem(68)<= x"ff";    --
BNE to 60
mem(75)<=x"8c";mem(74)<= x"a1";mem(73)<= x"a0";mem(72)<= x"01";    --
subCC i=1   rs=6  rd=6
mem(79)<=x"02";mem(78)<= x"bf";mem(77)<= x"ff";mem(76)<= x"f6";    --
BE to 36
mem(83)<=x"40";mem(82)<= x"00";mem(81)<= x"00";mem(80)<= x"0e";    --
CALL

-----------------------------------------------------------------

   s_1<=instr_addr+1;
   s_2<=instr_addr+2;
   s_3<=instr_addr+3;

instr_out(7 downto 0) <= mem(conv_integer(instr_addr)); -- read
instr_out(15 downto 8) <= mem(conv_integer(s_1)); -- read
instr_out(23 downto 16) <= mem(conv_integer(s_2)); -- read
instr_out(31 downto 24) <= mem(conv_integer(s_3)); -- read

end arc_rom;
```

--prog counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity prog_count is
  port(clk: in std_logic;
       reset:in std_logic;
       en_pc:in std_logic;
       pc_in: in unsigned(31 downto 0);
       pc_out: out unsigned(31 downto 0));
 end prog_count;

 architecture arc_prog_count of prog_count is
 begin

   process(clk)
     begin
     if(en_pc='1') then
      if(reset='1') then
         pc_out<=x"00000000";
      elsif(clk'event and clk='1') then
         pc_out<=pc_in;
      end if;
     end if;
```

```
   end process;

 end arc_prog_count;
```

--shift extend

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity shift_extend is
  port(a: in unsigned(21 downto 0);
       c: out unsigned(31 downto 0));
end shift_extend;

architecture arc_shift_extend of shift_extend is
begin

  c<=("11111111"&a&"00") when a(21)='1' else
     ("00000000"&a&"00");
end arc_shift_extend;
```

--sign extend

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity sign_extend is
  port(a: in unsigned(12 downto 0);
       c: out unsigned(31 downto 0));
end sign_extend;

architecture arc_sign_extend of sign_extend is
begin

  c<=("1111111111111111111"&a) when a(12)='1' else
     ("0000000000000000000"&a);

end arc_sign_extend;
```

--zero extend

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity zero_extend is
  port(a: in unsigned(4 downto 0);
       c: out unsigned(31 downto 0));
end zero_extend;

architecture arc_zero_extend of zero_extend is
begin
  c<=conv_unsigned(0,27)&a;
end zero_extend;
```

--mux 2X1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mux_2 is
  port(a: in unsigned(31 downto 0);
       b: in unsigned(31 downto 0);
       sel: in std_logic;
       c: out unsigned(31 downto 0));
end mux_2;

architecture arc_mux_2 of mux_2 is
begin

  c<=a when sel='0' else
     b;

end arc_mux_2;
```

--mux 4X1

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mux_4 is
  port(a: in unsigned(31 downto 0);
       b: in unsigned(31 downto 0);
       d: in unsigned(31 downto 0);
       e: in unsigned(31 downto 0);
       sel: in unsigned(1 downto 0);
       c: out unsigned(31 downto 0));
end mux_4;

architecture arc_mux_4 of mux_4 is
signal sel_int: integer range 0 to 3;
begin
  sel_int<=conv_integer(sel);
  c<=a when sel_int=0 else
     b when sel_int=1 else
     d when sel_int=2 else
     e;

end arc_mux_4;
```

--mux 8X1

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mux_8 is
  port(a: in unsigned(31 downto 0);
       b: in unsigned(31 downto 0);
       d: in unsigned(31 downto 0);
       e: in unsigned(31 downto 0);
       f: in unsigned(31 downto 0);
       g: in unsigned(31 downto 0);
       h: in unsigned(31 downto 0);
       k: in unsigned(31 downto 0);
       sel: in unsigned(2 downto 0);
       c: out unsigned(31 downto 0));
end mux_8;

architecture arc_mux_8 of mux_8 is
signal sel_int: integer range 0 to 7;
begin
  sel_int<=conv_integer(sel);
  c<=a when sel_int=0 else
     b when sel_int=1 else
     d when sel_int=2 else
     e when sel_int=3 else
     f when sel_int=4 else
     g when sel_int=5 else
     h when sel_int=6 else
     k;

end arc_mux_8;
```

--adder_4

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity add_4 is
  port(add_in: in unsigned(31 downto 0);
       add_out: out unsigned(31 downto 0));
 end add_4;

 architecture arc_add_4 of add_4 is
 begin

   add_out<=add_in + 4;

 end arc_add_4;
```

--reg_8

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity reg_8 is
  port(clk: in std_logic;
       add_1: in unsigned(4 downto 0);
       add_2: in unsigned(4 downto 0);
       add_3: in unsigned(4 downto 0);
       data in: in unsigned(31 downto 0);
       write_en: in std_logic;
       data_1: out unsigned(31 downto 0);
       data_2: out unsigned(31 downto 0);
       data_3: out unsigned(31 downto 0));

end reg_8;

architecture arc_reg_8 of reg_8 is
subtype vector is unsigned(31 downto 0);
type matrix is array(0 to 7) of vector;
signal reg: matrix :=(others=>(others=>'0'));
signal add_11: unsigned(2 downto 0);
signal add_22: unsigned(2 downto 0);
signal add_33: unsigned(2 downto 0);
signal s_1,s_2: std_logic;
signal s_3: integer range 0 to 7;
begin

  add_11<=add_1(2 downto 0);
  add_22<=add_2(2 downto 0);
  add_33<=add_3(2 downto 0);

  s_1<= not(add_3(4) or add_3(3));
  s_2<=s_1 and write_en;
  s_3<=conv_integer(add_33);


  data_1<=reg(conv_integer(add_11));
  data_2<=reg(conv_integer(add_22));
  data_3<=reg(conv_integer(add_33));

  process(clk,s_2)
    begin
      if(clk'event and clk='1') then
        if(s_2='1') then
          if(s_3=0) then
            reg(s_3)<=x"00000000";
          else reg(s_3)<=data_in;
          end if;
        end if;
      end if;
  end process;

end arc_reg_8;
```

--reg_32

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity reg_32 is
  port(clk: in std_logic;
       add_1: in unsigned(4 downto 0);
       add_2: in unsigned(4 downto 0);
       add_3: in unsigned(4 downto 0);
       data in: in unsigned(31 downto 0);
       write_en: in std_logic;
       current_window: in std_logic;
       data_1: out unsigned(31 downto 0);
       data_2: out unsigned(31 downto 0);
       data_3: out unsigned(31 downto 0));

end reg_32;

architecture arc_reg_32 of reg_32 is
subtype vector is unsigned(31 downto 0);
type matrix is array(0 to 31) of vector;
signal reg: matrix :=(others=>(others=>'0'));
signal add_11,add_12: unsigned(4 downto 0);
signal add_21,add_22: unsigned(4 downto 0);
signal add_31,add_32: unsigned(4 downto 0);
signal s_1,s_2: std_logic;
signal s_3: integer range 0 to 31;
begin

  s_1<=add_3(4) or add_3(3);
  s_2<=s_1 and write_en;
  s_3<=conv_integer(add_32);

  add_11<=add_1 - 8;
  add_12<=add_11 when current_window='1' else
          add_11 + 16;

  add_21<=add_2 - 8;
  add_22<=add_21 when current_window='1' else
          add_21 + 16;

  add_31<=add_3 - 8;
  add_32<=add_31 when current_window='1' else
          add_31 + 16;


  data_1<=reg(conv_integer(add_12));
  data_2<=reg(conv_integer(add_22));
  data_3<=reg(conv_integer(add_32));

  process(clk,s_2)
    begin
      if(clk'event and clk='1') then
        if(s_2='1') then
          reg(s_3)<=data_in;
        end if;
      end if;
  end process;
end arc_reg_32;
```

--register file

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity register_file is
  port(clk: in std_logic;
       addr_1: in unsigned(4 downto 0);
       addr_2: in unsigned(4 downto 0);
       addr_3: in unsigned(4 downto 0);
       data in: in unsigned(31 downto 0);
       write_en: in std_logic;
       current_window: in std_logic;
       data_1: out unsigned(31 downto 0);
       data_2: out unsigned(31 downto 0);
       data_3: out unsigned(31 downto 0));

end register_file;

architecture arc_register_file of register_file is
signal s_1,s_2,s_3: unsigned(31 downto 0);
signal s_4,s_5,s_6: unsigned(31 downto 0);
signal s_7,s_8,s_9: std_logic;
begin

  u1: entity work.reg_32(arc_reg_32) port
map(clk=>clk,add_1=>addr_1,add_2=>addr_2,add_3=>addr_3,data_in=>data_
in,
write_en=>write_en,current_window=>current_window,data_1=>s_1,data_2=
>s_2,data_3=>s_3);

  u2: entity work.reg_8(arc_reg_8) port
map(clk=>clk,add_1=>addr_1,add_2=>addr_2,add_3=>addr_3,data_in=>data_
in,
write_en=>write_en,data_1=>s_4,data_2=>s_5,data_3=>s_6);

  s_7<=addr_1(4) or addr_1(3);
  s_8<=addr_2(4) or addr_2(3);
  s_9<=addr_3(4) or addr_3(3);

  data_1<=s_1 when s_7='1' else
          s_4 when s_7='0';

  data_2<=s_2 when s_8='1' else
          s_5 when s_8='0';

  data_3<=s_3 when s_9='1' else
          s_6 when s_9='0';

end arc_register_file;
```

--status register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity status_register is
  port(clk: in std_logic;
       reset: in std_logic;
       en_icc: in std_logic;
       negative: in std_logic;
       zero: in std logic;
       over_flow: in std_logic;
       carry:in std_logic;
       borrow: in std_logic;
       addr_1: in unsigned(1 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       trap_type: in unsigned(7 downto 0);
       data_out,trap_register,window_invalid: out unsigned(31 downto
0);
       current_window: out std_logic;
       flags: out unsigned(3 downto 0));

end status_register;

architecture arc_status_register of status_register is
subtype vector is unsigned(31 downto 0);
type matrix is array(0 to 3) of vector;
signal reg: matrix :=(others=>(others=>'0'));
begin

  data_out<=reg(conv_integer(addr_1));
  window_invalid<=reg(1);
  trap_register<=reg(2);
  current_window<=reg(0)(0);
  flags(3 downto 0)<=reg(0)(23 downto 20);


process(clk,reset,negative,zero,over_flow,carry,borrow,trap_type,data
_in,en_icc)
    begin

     if(clk'event and clk='1') then
      if(reset='1') then
        reg(0)<=x"00000001";
        reg(1)(1 downto 0)<="11";
        reg(1)(31 downto 2)<=x"0000000"&"00";
        reg(2)(3 downto 0)<=x"0";
        reg(2)(31 downto 12)<=x"00000";
      else
         if(write en='1') then
            case addr_1 is
         when "00"=>
          reg(0)(13 downto 0)<=data_in(13 downto 0);
          reg(0)(19 downto 14)<="000000";
          reg(0)(31 downto 24)<=x"3a";

         when "01"=>
          reg(1)(1 downto 0)<="11";
          reg(1)(31 downto 2)<=x"0000000"&"00";
```

```
          when "10"=>
           reg(2)(3 downto 0)<="0000";
           reg(2)(31 downto 12)<=x"00000";

          when "11"=>
           reg(3)<=data_in;

          when others=>null;
          end case;
             end if;
          end if;
             end if;

              if(en_icc='1')then
                  reg(0)(20)<=(carry or borrow);
               reg(0)(21)<=over_flow ;
               reg(0)(22)<=zero ;
               reg(0)(23)<=negative ;
           end if;
          reg(2)(11 downto 4)<=trap_type;


  end process;

end arc_status_register;
```

--window trap

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity window_trap is
  port(a: in unsigned(4 downto 0);
       b: in unsigned(31 downto 0);
       en_trap: out std_logic);
  end window_trap;

architecture arc_window_trap of window_trap is

begin

    en_trap<=not b(conv_integer(a));

end arc_window_trap;
```

-------======CACHE========-----

--set0

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set0 is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std_logic;
       darity: in std_logic;
       data_out: out unsigned(31 downto 0);
       hit: out std_logic;
       least: out std_logic;
       d_0,d_1: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0));
end set0;

architecture arc_set0 of set0 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal_0,equal_1,hit_0,hit_1,s,least_disabled: std_logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin

  address_tag<=address(31 downto 7);
  block_offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);

  s<= (not (set_sel(0))) and  (not (set_sel(1))) and (not
(set_sel(2)));


     equal_0<='1' when address_tag=tag_0 else
              '0';

     equal_1<='1' when address_tag=tag_1 else
              '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit_0 or hit_1;


    least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                '0' when (s='1' and hit_0='1' and darity='1') else
                least_used;
```

```
mux_4_0:entity work.mux_4(arc_mux_4) port
map(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>bloc
k_offset,
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);

    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);


process(clk,write_en,address_tag,block_offset,data_in,hit_0,darity)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1' or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
              valid_1<=valid;
              d_1<=darity;

            end if;
          end if;
        end if;
        end if;
      end process;

      least_disabled<=least_used and (not darity);

      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set0;
```

--set1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set1 is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std_logic;
       darity: in std_logic;
       data_out: out unsigned(31 downto 0);
       hit: out std_logic;
       least: out std_logic;
```

```
       d_0,d_1: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0));
end set1;

architecture arc_set1 of set1 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal_0,equal_1,hit_0,hit_1,s,least_disabled: std_logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin
  address_tag<=address(31 downto 7);
  block_offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);

  s<= set_sel(0) and  (not (set_sel(1))) and (not (set_sel(2)));
  least_disabled<=least_used and (not darity);


     equal_0<='1' when address_tag=tag_0 else
              '0';

     equal_1<='1' when address_tag=tag_1 else
              '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit_0 or hit_1;


    least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                '0' when (s='1' and hit_0='1' and darity='1') else
                least_used;


    mux_4_0:entity work.mux_4(arc_mux_4)
portmap(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>
block_offset,
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);

    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);

    process(clk,write_en,address_tag,block_offset,data_in,hit_0)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
```

```
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1' or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
              valid_1<=valid;
              d_1<=darity;
            end if;
          end if;
        end if;
        end if;
      end process;

      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set1;
```

--set2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set2 is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std_logic;
       darity: in std_logic;
       data_out: out unsigned(31 downto 0);
       hit: out std_logic;
       least: out std_logic;
       d_0,d_1: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0));
end set2;

architecture arc_set2 of set2 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal 0,equal 1,hit 0,hit 1,s,least disabled: std logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin
  address_tag<=address(31 downto 7);
  block_offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);
```

```
  s<=(not (set_sel(0))) and set_sel(1) and (not (set_sel(2)));
  least_disabled<=least_used and (not darity);

     equal_0<='1' when address_tag=tag_0 else
               '0';

     equal_1<='1' when address_tag=tag_1 else
               '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit_0 or hit_1;


      least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                  '0' when (s='1' and hit_0='1' and darity='1') else
                  least_used;

    mux_4_0:entity work.mux_4(arc_mux_4) port
map(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>bloc
k_offset,
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);

    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);

    process(clk,write_en,address_tag,block_offset,data_in,hit_0)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1' or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
              valid_1<=valid;
              d_1<=darity;
            end if;
          end if;
        end if;
        end if;
      end process;

      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set2;
```

--set3

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set3 is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std logic;
       darity: in std_logic;
       data_out: out unsigned(31 downto 0);
       hit: out std_logic;
       least: out std_logic;
       d_0,d_1: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0));
end set3;

architecture arc_set3 of set3 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal_0,equal_1,hit_0,hit_1,s,least_disabled: std_logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin
  address_tag<=address(31 downto 7);
  block_offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);

  s<=set_sel(0) and set_sel(1) and (not (set_sel(2)));
  least_disabled<=least_used and (not darity);

     equal_0<='1' when address_tag=tag_0 else
              '0';

     equal_1<='1' when address_tag=tag_1 else
              '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit 0 or hit 1;


    least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                '0' when (s='1' and hit_0='1' and darity='1') else
                least_used;

    mux_4_0:entity work.mux_4(arc_mux_4) port
map(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>bloc
k_offset,
```

```
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);

    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);

    process(clk,write_en,address_tag,block_offset,data_in,hit_0)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1' or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
              valid_1<=valid;
              d_1<=darity;
            end if;
          end if;
        end if;
        end if;
      end process;

      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set3;
```

--set4

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set4 is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std_logic;
       darity: in std_logic;
       data out: out unsigned(31 downto 0);
       hit: out std_logic;
       least: out std_logic;
       d_0,d_1: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0));
end set4;

architecture arc_set4 of set4 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
```

```
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal_0,equal_1,hit_0,hit_1,s,least_disabled: std_logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin
  address_tag<=address(31 downto 7);
  block_offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);

  s<=(not (set_sel(0))) and (not (set_sel(1))) and set_sel(2);
  least_disabled<=least_used and (not darity);

     equal_0<='1' when address_tag=tag_0 else
              '0';

     equal_1<='1' when address_tag=tag_1 else
              '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit_0 or hit_1;


    least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                '0' when (s='1' and hit_0='1' and darity='1') else
                least_used;

    mux_4_0:entity work.mux_4(arc_mux_4) port
map(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>bloc
k_offset,
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);

    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);

    process(clk,write_en,address_tag,block_offset,data_in,hit_0)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1'or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
```

```
              valid_1<=valid;
              d_1<=darity;
            end if;
          end if;
        end if;
        end if;
      end process;

      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set4;
```

--set5

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set5 is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std_logic;
       darity: in std_logic;
       data_out: out unsigned(31 downto 0);
       hit: out std_logic;
       least: out std_logic;
       d_0,d_1: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0));
end set5;

architecture arc_set5 of set5 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal_0,equal_1,hit_0,hit_1,s,least_disabled: std_logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin
  address_tag<=address(31 downto 7);
  block offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);

  s<= set_sel(0) and  (not (set_sel(1))) and set_sel(2);
  least_disabled<=least_used and (not darity);

     equal_0<='1' when address_tag=tag_0 else
              '0';

     equal_1<='1' when address_tag=tag_1 else
```

```
              '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit_0 or hit_1;

      least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                 '0' when (s='1' and hit_0='1' and darity='1') else
                 least_used;


    mux_4_0:entity work.mux_4(arc_mux_4) port
map(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>bloc
k_offset,
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);

    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);

    process(clk,write_en,address_tag,block_offset,data_in,hit_0)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1' or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
              valid_1<=valid;
              d_1<=darity;
            end if;
          end if;
        end if;
        end if;
      end process;

      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set5;
```

--set6

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set6 is
  port(clk: in std_logic;
```

```
        address: in unsigned(31 downto 0);
        data_in: in unsigned(31 downto 0);
        write_en: in std_logic;
        valid: in std_logic;
        darity: in std_logic;
        data_out: out unsigned(31 downto 0);
        hit: out std_logic;
        least: out std_logic;
        d_0,d_1: out std_logic;
        tag0: out unsigned(24 downto 0);
        tag1: out unsigned(24 downto 0));
end set6;

architecture arc_set6 of set6 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal_0,equal_1,hit_0,hit_1,s,least_disabled: std_logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin
  address_tag<=address(31 downto 7);
  block_offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);

  s<= (not (set_sel(0))) and  set_sel(1) and set_sel(2);
  least_disabled<=least_used and (not darity);

     equal_0<='1' when address_tag=tag_0 else
              '0';

     equal_1<='1' when address_tag=tag_1 else
              '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit_0 or hit_1;


    least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                '0' when (s='1' and hit_0='1' and darity='1') else
                least_used;

    mux_4_0:entity work.mux_4(arc_mux_4) port
map(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>bloc
k_offset,
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);
```

```
    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);

    process(clk,write_en,address_tag,block_offset,data_in,hit_0)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1' or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
              valid_1<=valid;
              d_1<=darity;
            end if;
          end if;
        end if;
        end if;
      end process;

      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set6;
```

--set7

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity set7 is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std_logic;
       darity: in std_logic;
       data_out: out unsigned(31 downto 0);
       hit: out std_logic;
       least: out std_logic;
       d_0,d_1: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0));
end set7;

architecture arc_set7 of set7 is
type mem_type is array(0 to 3) of unsigned(31 downto 0);
signal block_0 : mem_type;
signal block_1 : mem_type;
signal tag_0 : unsigned(24 downto 0);
signal tag_1 : unsigned(24 downto 0);
signal address_tag: unsigned(24 downto 0);
signal equal_0,equal_1,hit_0,hit_1,s,least_disabled: std_logic:='0';
signal valid_0,valid_1,least_used: std_logic:='0';
```

```
signal data_0,data_1: unsigned(31 downto 0);
signal block_offset: unsigned(1 downto 0);
signal set_sel: unsigned(2 downto 0);

begin
  address_tag<=address(31 downto 7);
  block_offset<=address(3 downto 2);
  set_sel<=address(6 downto 4);

  s<= set_sel(0) and  set_sel(1) and set_sel(2);
  least_disabled<=least_used and (not darity);

     equal_0<='1' when address_tag=tag_0 else
              '0';

     equal_1<='1' when address_tag=tag_1 else
              '0';


    hit_0<=equal_0 and valid_0;
    hit_1<=equal_1 and valid_1;
    hit<=hit_0 or hit_1;


    least_used<='1' when (s='1' and hit_1='1' and darity='1') else
                '0' when (s='1' and hit_0='1' and darity='1') else
                least_used;

    mux_4_0:entity work.mux_4(arc_mux_4) port
map(a=>block_0(0),b=>block_0(1),d=>block_0(2),e=>block_0(3),sel=>bloc
k_offset,
c=>data_0);

    mux_4_1:entity work.mux_4(arc_mux_4) port
map(a=>block_1(0),b=>block_1(1),d=>block_1(2),e=>block_1(3),sel=>bloc
k_offset,
c=>data_1);

    mux_2_0:entity work.mux_2(arc_mux_2) port
map(a=>data_0,b=>data_1,sel=>hit_1,c=>data_out);

    process(clk,write_en,address_tag,block_offset,data_in,hit_0)
      begin
        if(clk'event and clk='1') then
          if(s='1')then
          if(write_en='1')then
            if(valid_0='0' or least_disabled='1' or hit_0='1')then
              block_0(conv_integer(block_offset))<=data_in;
              tag_0<=address_tag;
              valid_0<=valid;
              d_0<=darity;
            else
              block_1(conv_integer(block_offset))<=data_in;
              tag_1<=address_tag;
              valid_1<=valid;
              d_1<=darity;
            end if;
          end if;
        end if;
        end if;
      end process;
```

```
      tag0<=tag_0;
      tag1<=tag_1;
      least<=least_used;

  end arc_set7;
```

--cache

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity cache is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       data_in: in unsigned(31 downto 0);
       write_en: in std_logic;
       valid: in std_logic;
       darity: in std_logic;
       data_out: out unsigned(31 downto 0);
       hit: out std_logic;
       tag0: out unsigned(24 downto 0);
       tag1: out unsigned(24 downto 0);
       least_used: out std_logic;
       d_0,d_1: out std_logic);
       end cache;

architecture arc_cache of cache is
signal data_0,data_1,data_2,data_3,data_4,data_5,data_6,data_7:
unsigned(31 downto 0);

signal hit_0,hit_1,hit_2,hit_3,hit_4,hit_5,hit_6,hit_7: std_logic;

signal
least_0,least_1,least_2,least_3,least_4,least_5,least_6,least_7:
std_logic;

signal
d0_0,d1_0,d0_1,d1_1,d0_2,d1_2,d0_3,d1_3,d0_4,d1_4,d0_5,d1_5,d0_6,d1_6
,d0_7,d1_7: std_logic;

signaltag0_0,tag1_0,tag0_1,tag1_1,tag0_2,tag1_2,tag0_3,tag1_3,tag0_4,
tag1_4,tag0_5,tag1_5,tag0_6,tag1_6,tag0_7,tag1_7: unsigned(24 downto
0);

signal set_sel: unsigned(2 downto 0);
begin

   set_sel<=address(6 downto 4);

   set_0: entity work.set0(arc_set0) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_0,hit=>hit_0,tag0=>tag0_0,tag1=>tag1_
0,least=>least_0,
d_0=>d0_0,d_1=>d1_0);
```

```
   set_1: entity work.set1(arc_set1) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_1,hit=>hit_1,tag0=>tag0_1,tag1=>tag1_
1,least=>least_1,
d_0=>d0_1,d_1=>d1_1);

   set_2: entity work.set2(arc_set2) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_2,hit=>hit_2,tag0=>tag0_2,tag1=>tag1_
2,least=>least_2,
d_0=>d0_2,d_1=>d1_2);

   set_3: entity work.set3(arc_set3) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_3,hit=>hit_3,tag0=>tag0_3,tag1=>tag1_
3,least=>least_3,
d_0=>d0_3,d_1=>d1_3);

   set_4: entity work.set4(arc_set4) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_4,hit=>hit_4,tag0=>tag0_4,tag1=>tag1_
4,least=>least_4,
d_0=>d0_4,d_1=>d1_4);

   set_5: entity work.set5(arc_set5) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_5,hit=>hit_5,tag0=>tag0_5,tag1=>tag1_
5,least=>least_5,
d_0=>d0_5,d_1=>d1_5);

   set_6: entity work.set6(arc_set6) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_6,hit=>hit_6,tag0=>tag0_6,tag1=>tag1_
6,least=>least_6,
d_0=>d0_6,d_1=>d1_6);

   set_7: entity work.set7(arc_set7) port
map(clk=>clk,address=>address,write_en=>write_en,valid=>valid,darity=
>darity,
data_in=>data_in,data_out=>data_7,hit=>hit_7,tag0=>tag0_7,tag1=>tag1_
7,least=>least_7,
d_0=>d0_7,d_1=>d1_7);

  ------------------------------------------------------------------
------------------
  with set_sel select
  data_out<=data_0 when "000",
            data_1 when "001",
            data_2 when "010",
            data_3 when "011",
            data_4 when "100",
            data_5 when "101",
            data_6 when "110",
            data_7 when others;
```

```
with set_sel select
     hit<=hit_0 when "000",
          hit_1 when "001",
          hit_2 when "010",
          hit_3 when "011",
          hit_4 when "100",
          hit_5 when "101",
          hit_6 when "110",
          hit_7 when others;

with set_sel select
     tag0<=tag0_0 when "000",
          tag0_1 when "001",
          tag0_2 when "010",
          tag0_3 when "011",
          tag0_4 when "100",
          tag0_5 when "101",
          tag0_6 when "110",
          tag0_7 when others;

with set_sel select
     tag1<=tag1_0 when "000",
          tag1_1 when "001",
          tag1_2 when "010",
          tag1_3 when "011",
          tag1_4 when "100",
          tag1_5 when "101",
          tag1_6 when "110",
          tag1_7 when others;

with set_sel select
     least_used<=least_0 when "000",
          least_1 when "001",
          least_2 when "010",
          least_3 when "011",
          least_4 when "100",
          least_5 when "101",
          least_6 when "110",
          least_7 when others;

with set_sel select
     d_0<=d0_0 when "000",
          d0_1 when "001",
          d0_2 when "010",
          d0_3 when "011",
          d0_4 when "100",
          d0_5 when "101",
          d0_6 when "110",
          d0_7 when others;

with set_sel select
     d_1<=d1_0 when "000",
          d1_1 when "001",
          d1_2 when "010",
          d1_3 when "011",
          d1_4 when "100",
          d1_5 when "101",
          d1_6 when "110",
          d1_7 when others;
```

```
end arc_cache;
```

--Cache controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity cache_controller is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       tag0: in unsigned(24 downto 0);
       tag1: in unsigned(24 downto 0);
       enable: in std_logic;
       write_en: in std_logic;
       least_used: in std_logic;
       d_0,d_1: in std_logic;
       internal_write_en: out std_logic;
       valid: out std_logic;
       darity: out std_logic;
       done: out std_logic;
       external_write_en: out std_logic;
       address_out: out unsigned(31 downto 0));

end cache_controller;

architecture arc_cache_controller of cache_controller is
type state
is(state0,state1,state2,state3,state4,state5,state6,state7,state8);
signal pr_state,nx_state: state;
signal reset: std_logic;
signal latency: integer range 0 to 7;
signal tag_bit: std_logic;
begin

  process(clk)
   begin
    if(clk'event and clk='1')then
      pr_state<=nx_state;
    end if;
  end process;

  process(clk,enable,address,write_en)
   begin
     if(clk'event and clk='1')then
       latency<=latency+1;
       if(latency=4 or reset='1')then
         latency<=0;
       end if;
     end if;

     case pr state is
    ---------------------------------------- idle state
       when state0=>
         external_write_en<='0';
         done<='0';
         address_out<=address;
         if(enable='1')then
           internal_write_en<='0';
           valid<='0';
           darity<='0';
```

```
        reset<='0';
        if(least_used='1')then
          if(d_0='1')then
            nx_state<=state5;
            tag_bit<='0';
          else nx_state<=state1;
          end if;
        else
          if(d_1='1')then
            nx_state<=state5;
            tag_bit<='1';
          else nx_state<=state1;
          end if;
        end if;

      else   reset<='1';
             nx_state<=state0;
             internal_write_en<=write_en;
             valid<=write_en;
             darity<='1';

      end if;

------------------------------------ miss and writing

    when state1=>
      address_out<=address(31 downto 4)&"00"&address(1 downto 0);
      if(latency=4)then
        internal_write_en<='1';
        nx_state<=state2;
      else  nx_state<=state1;
            internal_write_en<='0';
      end if;

    when state2=>
      address_out<=address(31 downto 4)&"01"&address(1 downto 0);
      if(latency=4)then
        internal_write_en<='1';
        nx_state<=state3;
      else  nx_state<=state2;
            internal_write_en<='0';
      end if;

    when state3=>
      address_out<=address(31 downto 4)&"10"&address(1 downto 0);
      if(latency=4)then
        internal_write_en<='1';
        nx_state<=state4;
      else  nx_state<=state3;
            internal_write_en<='0';
      end if;

    when state4=>
      address_out<=address(31 downto 4)&"11"&address(1 downto 0);
      if(latency=4)then
        internal_write_en<='1';
        valid<='1';
        nx_state<=state0;
        done<='1';
      else  nx_state<=state4;
            internal_write_en<='0';
```

```
              valid<='0';
              done<='0';
          end if;


     ------------------------------------------ darity and write back
         when state5=>

           if(tag_bit='1')then
              address_out<=tag1&address(6 downto 4)&"00"&address(1
downto 0);
            else address_out<=tag0&address(6 downto 4)&"00"&address(1
downto 0);
            end if;

          if(latency=2)then
            external_write_en<='1';
            nx_state<=state6;
            reset<='1';
          else  external_write_en<='0';
            nx_state<=state5;
            reset<='0';
          end if;

         when state6=>
           if(tag_bit='1')then
              address_out<=tag1&address(6 downto 4)&"01"&address(1
downto 0);
            else address_out<=tag0&address(6 downto 4)&"01"&address(1
downto 0);
            end if;

          if(latency=2)then
            external_write_en<='1';
            nx_state<=state7;
            reset<='1';
          else  external_write_en<='0';
            nx_state<=state6;
            reset<='0';
          end if;

          when state7=>
           if(tag_bit='1')then
              address_out<=tag1&address(6 downto 4)&"10"&address(1
downto 0);
            else address_out<=tag0&address(6 downto 4)&"10"&address(1
downto 0);
            end if;

          if(latency=2)then
            external_write_en<='1';
            nx_state<=state8;
            reset<='1';
          else  external_write_en<='0';
            nx_state<=state7;
            reset<='0';
          end if;

          when state8=>
           if(tag_bit='1')then
```

```
            address_out<=tag1&address(6 downto 4)&"11"&address(1
downto 0);
          else address_out<=tag0&address(6 downto 4)&"11"&address(1
downto 0);
          end if;

         if(latency=2)then
           external_write_en<='1';
           nx_state<=state1;
           reset<='1';
         else  external_write_en<='0';
           nx_state<=state8;
           reset<='0';
         end if;

       end case;
     end process;

   end arc_cache_controller;
```

--Data cache

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity data_cache is
  port(clk: in std_logic;
       address: in unsigned(31 downto 0);
       enable: in std_logic;
       write_en: in std_logic;
       data_in: in unsigned(31 downto 0);
       hit: out std_logic;
       external_write_en: out std_logic;
       address_out: out unsigned(31 downto 0);
       data_out: out unsigned(31 downto 0);
       done: out std_logic);

end data_cache;

architecture arc_data_cache of data_cache is
signal addr_out: unsigned(31 downto 0);
signal tag0: unsigned(24 downto 0);
signal tag1: unsigned(24 downto 0);
signal internal_write_en: std_logic;
signal valid: std_logic;
signal darity: std_logic;
signal least_used: std_logic;
signal d_0,d_1: std_logic;
begin

  cach_cont: entity work.cache_controller(arc_cache_controller) port
map(clk=>clk,address=>address,tag0=>tag0,tag1=>tag1,enable=>enable,wr
ite_en=>write_en,
least_used=>least_used,d_0=>d_0,d_1=>d_1,valid=>valid,darity=>darity,
internal_write_en=>internal_write_en,address_out=>addr_out,
external_write_en=>external_write_en,done=>done);

  cach: entity work.cache(arc_cache)
```

```
port
map(clk=>clk,address=>addr_out,write_en=>internal_write_en,data_in=>d
ata_in,
valid=>valid,darity=>darity,data_out=>data_out,tag0=>tag0,tag1=>tag1,
hit=>hit,
least_used=>least_used,d_0=>d_0,d_1=>d_1);

  address_out<=addr_out;
end arc_data_cache;
```

--Data Path

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity data_path is
  port(clk: in std_logic;
       reset: in std_logic;
       cache_en: in std_logic;
       data_in: in unsigned(31 downto 0);
       pc_intrr: in unsigned(31 downto 0);
       write_en_reg: in std_logic;
       write_en_st: in std_logic;
       write_en_cache: in std_logic;
       en_pc: in std_logic;
       en_icc: in std_logic;
       alu_operation: in unsigned(3 downto 0);
       sel_main: in std_logic;
       sel_pc: in unsigned(2 downto 0);
       sel_reg: in unsigned(1 downto 0);
       sel_A: in std_logic;
       sel_B: in unsigned(1 downto 0);
       sel_cache_in: in std_logic;
       trap_type: in unsigned(7 downto 0);
       write_out: in std_logic;
       db_switches: in unsigned(31 downto 0);
       op: out unsigned(1 downto 0);
       op_2: out unsigned(2 downto 0);
       op_3: out unsigned(5 downto 0);
       cond: out unsigned(3 downto 0);
       i: out std_logic;
       flags: out unsigned(3 downto 0);
       en_trap: out std_logic;
       address: out unsigned(31 downto 0);
       processor_out:out unsigned(7 downto 0);
       hit: out std_logic;
       write_en_mem: out std_logic;
       data_out: out unsigned(31 downto 0);
       done: out std_logic);
 end data_path;

architecture arc_data_path of data_path is

signal pc_in,cache_in,cache_out: unsigned(31 downto 0);
signal pc_out,s: unsigned(31 downto 0);
signal addr_1: unsigned(4 downto 0);
signal addr_2: unsigned(4 downto 0);
signal addr_3: unsigned(4 downto 0);
signal data_in_reg: unsigned(31 downto 0);
signal data_1: unsigned(31 downto 0);
```

```
signal current_window: std_logic;
signal alu_in_0,alu_in_1,alu_in_2: unsigned(31 downto 0);
signal alu_out,alu_A,alu_B,main_path: unsigned(31 downto 0);
signal data_out_st: unsigned(31 downto 0);
signal pc_in_0,pc_in_1,pc_in_2,pc_in_3 : unsigned(31 downto 0);
signal shift_extend_out,trap_register,window_invalid: unsigned(31
downto 0);
signal negative: std_logic;
signal zero: std_logic;
signal over_flow: std_logic;
signal carry: std_logic;
signal borrow: std_logic;
signal data_out_reg: unsigned(31 downto 0);
begin

 op<=s(31 downto 30);
 op_2<=s(24 downto 22);
 op_3<=s(24 downto 19);
 cond<=s(28 downto 25);
 i<=s(13);
 addr_1<=s(18 downto 14);
 addr_2<=s(4 downto 0);
 addr_3<=s(29 downto 25);

 prog_count: entity work.prog_count(arc_prog_count) port
map(clk=>clk,reset=>reset,en_pc=>en_pc,pc_in=>pc_in,
 pc_out=>pc_out);

 add_4: entity work.add_4(arc_add_4) port
map(add_in=>pc_out,add_out=>pc_in_0);

 rom: entity work.rom(arc_rom) port map(instr_addr=>pc_out(7 downto
0),instr_out=>s);

 zero_extend_27: entity work.zero_extend_27(arc_zero_extend_27) port
map(a=>addr_2,c=>alu_in_1);

 sign_extend: entity work.sign_extend(arc_sign_extend) port
map(a=>s(12 downto 0),c=>alu_in_2);

 shift_extend: entity work.shift_extend(arc_shift_extend) port
map(a=>s(21 downto 0),c=>shift_extend_out);

 shift_2: entity work.shift_by_2(arc_shift_by_2) port map(a=>s(29
downto 0),c=>pc_in_2);

 adder: entity work.adder(arc_adder) port
map(a=>pc_out,b=>shift_extend_out,c=>pc_in_1);

 mux_2_0: entity work.mux_2(arc_mux_2) port
map(a=>data_1,b=>data_out_st,sel=>sel_A,c=>alu_A);

 mux_2_1: entity work.mux_2(arc_mux_2) port
map(a=>data_out_reg,b=>data_in,sel=>sel_cache_in,c=>cache_in);

 mux_2_2: entity work.mux_2(arc_mux_2) port
map(a=>alu_out,b=>cache_out,sel=>sel_main,c=>main_path);

 mux_4_0: entity work.mux_4(arc_mux_4) port
map(a=>main_path,b=>pc_out,d=>data_out_st,e=>db_switches,
  sel=>sel_reg,c=>data_in_reg);
```

```
 mux_4_1: entity work.mux_4(arc_mux_4) port
map(a=>alu_in_0,b=>alu_in_1,d=>alu_in_2,e=>x"00000000",
          sel=>sel_B,c=>alu_B);

 mux_8: entity work.mux_8(arc_mux_8) port
map(a=>pc_in_0,b=>pc_in_1,d=>pc_in_2,e=>main_path,f=>trap_register,

g=>pc_intrr,h=>x"00000000",k=>x"00000000",sel=>sel_pc,c=>pc_in);

 reg_file:entity work.register_file_1(arc_register_file_1) port
map(clk=>clk,addr_1=>addr_1,addr_2=>addr_2,addr_3=>addr_3,

data_in=>data_in_reg,write_en=>write_en_reg,current_window=>current_w
indow,data_1=>data_1,data_2=>alu_in_0,
 data_3=>data_out_reg);

 status_register:entity work.status_register(arc_status_register)
port map(clk=>clk,reset=>reset,
 addr_1=>addr_1(1 downto
0),data_in=>data_1,en_icc=>en_icc,write_en=>write_en_st,negative=>neg
ative,zero=>zero,

over_flow=>over_flow,carry=>carry,borrow=>borrow,trap_type=>trap_type
,current_window=>current_window,flags=>flags,

data_out=>data_out_st,trap_register=>trap_register,window_invalid=>wi
ndow_invalid);

 window_trap: entity work.window_trap(arc_window_trap) port
map(a=>alu_out(4 downto 0),b=>window_invalid,
 en_trap=>en_trap);

 alu: entity work.alu(arc_alu) port
map(a=>alu_A,b=>alu_B,sel=>alu_operation,c=>alu_out,negative=>negativ
e,
      zero=>zero,over_flow=>over_flow,carry=>carry,borrow=>borrow);

 data_cache: entity work.data_cache(arc_data_cache) port
map(clk=>clk,address=>alu_out,enable=>cache_en,

write_en=>write_en_cache,data_in=>cache_in,hit=>hit,external_write_en
=>write_en_mem,address_out=>address,
 data_out=>cache_out,done=>done);

 data_out<=cache_out;

   process(clk)
    begin
       if(clk'event and clk='1')then
         if(write_out='1')then
           processor_out<=data_out_reg(7 downto 0);
         end if;
       end if;
   end process;

 end arc_data_path;
```

--control unit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity control_unit is
  port(clk: in std_logic;
       op: in unsigned(1 downto 0);
       op_2: in unsigned(2 downto 0);
       op_3: in unsigned(5 downto 0);
       cond: in unsigned(3 downto 0);
       i: in std logic;
       flags: in unsigned(3 downto 0);
       en_trap: in std_logic;
       hit: in std_logic;
       done: in std_logic;
       interrupt: in std_logic;
       pc_intrr: out unsigned(31 downto 0);
       sel_pc: out unsigned(2 downto 0);
       en_pc: out std_logic;
           en_icc: out std_logic;
       write_en_reg: out std_logic;
       write_en_st: out std_logic;
           write_en_cache: out std_logic;
           write_out: out std_logic;
           cache_en: out std_logic;
       sel_reg: out unsigned(1 downto 0);
           sel_A: out std_logic;
       sel_B: out unsigned(1 downto 0);
       alu_operation: out unsigned(3 downto 0);
       sel_main: out std_logic;
       sel_cache_in: out std_logic;
       trap_type: out unsigned(7 downto 0):=x"00"
       );
end control_unit;


architecture arc_control_unit of control_unit is
type state is(normal,cache_operation);
signal pr_state,nx_state: state;
begin

    process(clk)
     begin
      if(clk'event and clk='1')then
        pr_state<=nx_state;
      end if;
    end process;

    process(pr_state,op,op_2,op_3,cond,i,flags,en_trap,hit,done)
    begin

   case pr_state is
    when normal=>
    if(interrupt='1')then
      pc_intrr<=x"00000030";
      sel_pc<="101";
      en_pc<='1';
      en_icc<='0';
      write_out<='0';
    else
```

```
    case op is
when "11"=>                     --FORMAT 3
case op_3 is
  when "000000"=>               --LD
    if(hit='1')then
    nx_state<=normal;
    sel_pc<="000";
    en_pc<='1';
    en_icc<='0';
    write_out<='0';
    write_en_reg<='1';
    write_en_st<='0';
    write_en_cache<='0';
    cache_en<='0';
    sel_reg<="00";
          sel_A<='0';
    if(i='1') then
    sel_B<="10";
    else sel_B<="00";
    end if;
    alu_operation<="0000";
    sel_cache_in<='0';
    sel_main<='1';

  else nx_state<=cache_operation;
    sel_pc<="000";
    en_pc<='0';
    en_icc<='0';
    write_out<='0';
    write_en_reg<='0';
    write_en_st<='0';
    write_en_cache<='0';
    cache_en<='1';
    sel_reg<="00";
          sel_A<='0';
    if(i='1') then
    sel_B<="10";
    else sel_B<="00";
    end if;
    alu_operation<="0000";
    sel_cache_in<='1';
    sel_main<='1';
  end if;



  when "000100"=>               --ST
  if(hit='1')then
      nx_state<=normal;
    sel_pc<="000";
    en_pc<='1';
    en_icc<='0';
    write_out<='0';
    write_en_reg<='0';
    write_en_st<='0';
    write_en_cache<='1';
    cache_en<='0';
    write_out<='0';
    sel_reg<="00";
          sel_A<='0';
    if(i='0')then
```

```
  sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

else nx_state<=cache_operation;
  sel_pc<="000";
  en_pc<='0';
  en_icc<='0';
  write_out<='0';
  write_en_reg<='0';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='1';
  sel_reg<="00";
        sel_A<='0';
  if(i='1') then
  sel_B<="10";
  else sel_B<="00";
  end if;
  alu_operation<="0000";
  sel_cache_in<='1';
  sel_main<='1';
end if;

when "010000"=>              --LDA
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
  write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="11";
        sel_A<='0';
  if(i='1') then
  sel_B<="10";
  else sel_B<="00";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='1';

  when "010100"=>              --STA
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
  write_en_reg<='0';
  write_en_st<='0';
  write_en_cache<='0';
  write_out<='1';
  cache_en<='0';
  sel_reg<="11";
        sel_A<='0';
  if(i='1') then
  sel_B<="10";
  else sel_B<="00";
  end if;
```

```
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='1';

when "001111"=>              --SWAP
if(hit='1')then
    nx_state<=normal;
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
  write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='1';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
  sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='1';

else nx_state<=cache_operation;
  sel_pc<="000";
  en_pc<='0';
  en_icc<='0';
  write_out<='0';
  write_en_reg<='0';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='1';
  sel_reg<="00";
        sel_A<='0';
  if(i='1') then
  sel_B<="10";
  else sel_B<="00";
  end if;
  alu_operation<="0000";
  sel_cache_in<='1';
  sel_main<='1';
end if;

when others=>
        sel_pc<="000";
    en_pc<='1';
    en_icc<='0';
  write_out<='0';
    write_en_reg<='0';
    write_en_st<='0';
    write_en_cache<='0';
    cache_en<='0';
            write_out<='0';
    sel_reg<="00";
          sel_A<='0';
    if(i='0')then
      sel_B<="00";
    else sel_B<="10";
    end if;
```

```
    alu_operation<="0000";
    sel_cache_in<='0';
    sel_main<='0';

end case;

when "10"=>                                --FORMAT 2
case op_3 is
  when "000001"=>                         -- AND
    sel_pc<="000";
    en_pc<='1';
    en_icc<='0';
  write_out<='0';
    write_en_reg<='1';
    write_en_st<='0';
    write_en_cache<='0';
    cache_en<='0';
    sel_reg<="00";
          sel_A<='0';
    if(i='0')then
      sel_B<="00";
    else sel_B<="10";
    end if;
    alu_operation<="1000";
    sel_cache_in<='0';
    sel_main<='0';

    when "010001"=>                     --ANDCC
    sel_pc<="000";
    en_icc<='1';
    write_out<='0';
    write_en_reg<='1';
    write_en_st<='0';
    write_en_cache<='0';
    cache_en<='0';
    sel_reg<="00";
          sel_A<='0';
    if(i='0')then
      sel_B<="00";
    else sel_B<="10";
    end if;
    alu_operation<="1000";
    sel_cache_in<='0';
    sel_main<='0';

    when "000101"=>                   --NAND
    sel_pc<="000";
    en_pc<='1';
    en_icc<='0';
  write_out<='0';
    write_en_reg<='1';
    write_en_st<='0';
    write_en_cache<='0';
    cache_en<='0';
    sel_reg<="00";
          sel_A<='0';
    if(i='0')then
      sel_B<="00";
    else sel_B<="10";
    end if;
    alu_operation<="1011";
```

```
     sel_cache_in<='0';
sel_main<='0';

     when "010101"=>              --NANDCC
     sel_pc<="000";
     en_icc<='1';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1011";
     sel_cache_in<='0';
sel_main<='0';

     when "000010"=>              --OR
     sel_pc<="000";
     en_pc<='1';
   en_icc<='0';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
     sel_reg<="00";
           sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1001";
     sel_cache_in<='0';
sel_main<='0';

     when "010010"=>              --ORCC
     sel_pc<="000";
     en_icc<='1';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1001";
     sel_cache_in<='0';
sel_main<='0';

     when "000110"=>              --NOR
     sel_pc<="000";
     en_pc<='1';
```

```
   en_icc<='0';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1100";
     sel_cache_in<='0';
sel_main<='0';

     when "010110"=>              --NORCC
     sel_pc<="000";
     en_icc<='1';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1100";
     sel_cache_in<='0';
sel_main<='0';

     when "000011"=>              --XOR
     sel_pc<="000";
     en_pc<='1';
   en_icc<='0';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1010";
     sel_cache_in<='0';
sel_main<='0';

     when "010011"=>              --XORCC
     sel_pc<="000";
     en_icc<='1';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
```

```
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1010";
     sel_cache_in<='0';
sel_main<='0';

     when "000111"=>              --XNOR
     sel_pc<="000";
     en_pc<='1';
   en_icc<='0';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
   cache_en<='0';
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1101";
     sel_cache_in<='0';
sel_main<='0';

     when "010111"=>              --XNORCC
     sel_pc<="000";
     en_icc<='1';
   write_out<='0';
     write_en_reg<='1';
     write_en_cache<='0';
   cache_en<='0';
     write_en_st<='0';
     sel_reg<="00";
        sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="10";
     end if;
     alu_operation<="1101";
     sel_cache_in<='0';
sel_main<='0';

     when "100101"=>              --SLL
     sel_pc<="000";
     en_pc<='1';
     en_icc<='0';
   write_out<='0';
     write_en_reg<='1';
     write_en_st<='0';
     write_en_cache<='0';
     cache_en<='0';
     sel_reg<="00";
           sel_A<='0';
     if(i='0')then
       sel_B<="00";
     else sel_B<="01";
```

```
      end if;
      alu_operation<="0010";
      sel_cache_in<='0';
      sel_main<='0';

      when "100110"=>               --SRL
      sel_pc<="000";
      en_pc<='1';
    en_icc<='0';
    write_out<='0';
      write_en_reg<='1';
      write_en_st<='0';
      write_en_cache<='0';
    cache_en<='0';
      sel_reg<="00";
         sel_A<='0';
      if(i='0')then
        sel_B<="00";
      else sel_B<="01";
      end if;
      alu_operation<="0011";
      sel_cache_in<='0';
sel_main<='0';

      when "100111"=>               --SRA
      sel_pc<="000";
      en_pc<='1';
    en_icc<='0';
    write_out<='0';
      write_en_reg<='1';
      write_en_st<='0';
      write_en_cache<='0';
    cache_en<='0';
      sel_reg<="00";
         sel_A<='0';
      if(i='0')then
        sel_B<="00";
      else sel_B<="01";
      end if;
      alu_operation<="0100";
      sel_cache_in<='0';
sel_main<='0';

      when "000000"=>               --ADD
      sel_pc<="000";
      en_pc<='1';
      en_icc<='0';
    write_out<='0';
      write_en_reg<='1';
      write_en_st<='0';
      write_en_cache<='0';
      cache_en<='0';
      sel_reg<="00";
            sel_A<='0';
      if(i='0')then
        sel_B<="00";
      else sel_B<="10";
      end if;
      alu_operation<="0000";
      sel_cache_in<='0';
sel_main<='0';
```

```
  when "010000"=>              --ADDCC
  sel_pc<="000";
  en_icc<='1';
write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "000100"=>              --SUB
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0001";
  sel_cache_in<='0';
  sel_main<='0';

  when "010100"=>              --SUBCC
  sel_pc<="000";
  en_icc<='1';
write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0001";
  sel_cache_in<='0';
  sel_main<='0';

  when "111100"=>              --SAVE
  if(en_trap='1') then
    sel_pc<="100";
    write_en_st<='0';
  else sel_pc<="000";
```

```
        write_en_st<='1';
    end if;
    en_pc<='1';
    en_icc<='0';
  write_out<='0';
    write_en_reg<='0';
    write_en_cache<='0';
    cache_en<='0';
    sel_reg<="00";
          sel_A<='1';
    if(i='0')then
      sel_B<="00";
    else sel_B<="10";
    end if;
    alu_operation<="0001";
    sel_cache_in<='0';
    sel_main<='0';
    trap_type<=x"00";

    when "111101"=>               --RESTORE
    if(en_trap='1') then
      sel_pc<="100";
      write_en_st<='0';
    else sel_pc<="000";
         write_en_st<='1';
    end if;
    en_pc<='1';
    en_icc<='0';
  write_out<='0';
    write_en_reg<='0';
    write_en_cache<='0';
    cache_en<='0';
    sel_reg<="00";
          sel_A<='1';
    if(i='0')then
      sel_B<="00";
    else sel_B<="10";
    end if;
    alu_operation<="0000";
    sel_cache_in<='0';
    sel_main<='0';
    trap_type<=x"00";

    when "111000"=>               --JUMP
    sel_pc<="011";
    en_pc<='1';
    en_icc<='0';
  write_out<='0';
    write_en_reg<='1';
    write_en_st<='0';
    write_en_cache<='0';
    cache_en<='0';
    sel_reg<="01";
            sel_A<='0';
    if(i='0')then
      sel_B<="00";
    else sel_B<="10";
    end if;
    alu_operation<="0000";
    sel_cache_in<='0';
    sel_main<='0';
```

```
  when "101000"=>              --RDASR
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="10";
          sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "101001"=>              --RDPSR
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="10";
          sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "101010"=>              --RDWIM
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="10";
          sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "101011"=>              --RDTBR
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
```

```
write_out<='0';
  write_en_reg<='1';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="10";
          sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "110000"=>               --WRASR
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='1';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "110001"=>               --WRPSR
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='1';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "110010"=>               --WRWIM
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='1';
  write_en_cache<='0';
```

```
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when "110011"=>              --WRTBR
  sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='1';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';

  when others=>
          sel_pc<="000";
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='1';
  write_en_cache<='0';
  cache_en<='0';
  sel_reg<="00";
        sel_A<='0';
  if(i='0')then
    sel_B<="00";
  else sel_B<="10";
  end if;
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='0';
  end case;

when "01"=>                   --FORMAT 1
sel_pc<="010";              --CALL
en_pc<='1';
en_icc<='0';
write_out<='0';
      write_en_reg<='0';
write_en_st<='0';
write_en_cache<='0';
cache_en<='0';
      sel_reg<="00";
      sel_A<='0';
```

```
      if(i='1') then
sel_B<="01";
else sel_B<="10";
end if;
alu_operation<="0000";
sel_cache_in<='0';
sel_main<='1';

when "00"=>                          --FORMAT 0
case op_2 is
when "100"=>                         --NOP
sel_pc<="000";
en_pc<='1';
en_icc<='0';
write_out<='0';
write_en_reg<='0';
write_en_st<='0';
write_en_cache<='0';
cache_en<='0';
      sel_reg<="00";
      sel_A<='0';
sel_B<="00";
alu_operation<="0000";
sel_cache_in<='0';
sel_main<='1';

when "010"=>
case cond is
  when "1001"=>                      --Branch on not equal
  if(flags(2)='0') then
   sel_pc<="001";
  else sel_pc<="000";
  end if;
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
          sel_reg<="00";
        sel_A<='0';
        sel_B<="00";
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='1';

  when "0001"=>                      --Branch on equal
  if(flags(2)='1') then
   sel_pc<="001";
  else sel_pc<="000";
  end if;
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
          sel_reg<="00";
        sel_A<='0';
```

```
  sel_B<="00";
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='1';

  when "1010"=>                    --Branch on GREATER
  if(not (flags(2) or (flags(3) xor flags(1)))='1') then
   sel_pc<="001";
  else sel_pc<="000";
  end if;
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
          sel_reg<="00";
        sel_A<='0';
  sel_B<="00";
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='1';

      when "0101"=>                  --Branch on carry set
  if(flags(0)='1') then
   sel_pc<="001";
  else sel_pc<="000";
  end if;
  en_pc<='1';
  en_icc<='0';
write_out<='0';
  write_en_reg<='0';
  write_en_st<='0';
  write_en_cache<='0';
  cache_en<='0';
          sel_reg<="00";
        sel_A<='0';
  sel_B<="00";
  alu_operation<="0000";
  sel_cache_in<='0';
  sel_main<='1';

when others=>
      sel_pc<="000";
en_pc<='1';
en_icc<='0';
write_out<='0';
write_en_reg<='1';
write_en_st<='0';
write_en_cache<='0';
cache_en<='0';
sel_reg<="00";
      sel_A<='0';
sel_B<="01";
alu_operation<="0000";
sel_cache_in<='0';
sel_main<='1';
end case;

when others=>
```

```
            sel_pc<="000";
        en_pc<='1';
        en_icc<='0';
        write_out<='0';
        write_en_reg<='1';
        write_en_st<='0';
        write_en_cache<='0';
        cache_en<='0';
        sel_reg<="00";
              sel_A<='0';
        sel_B<="00";
        alu_operation<="0000";
        sel_cache_in<='0';
        sel_main<='1';
        end case;


  when others=>
          sel_pc<="000";
        en_pc<='1';
        en_icc<='0';
        write_out<='0';
        write_en_reg<='1';
        write_en_st<='0';
        write_en_cache<='0';
        cache_en<='0';
        sel_reg<="00";
              sel_A<='0';
              if(i='1') then
        sel_B<="01";
        else sel_B<="10";
        end if;
        alu_operation<="0000";
        sel_cache_in<='0';
        sel_main<='1';
  end case;
end if;

  when cache_operation=>
    if(done='1')then
      nx_state<=normal;
    else
      nx_state<=cache_operation;
      sel_pc<="000";
      en_pc<='0';
      en_icc<='0';
      write_out<='0';
      write_en_reg<='0';
      write_en_st<='0';
      write_en_cache<='1';
      cache_en<='1';
      sel_reg<="00";
            sel_A<='0';
            if(i='1') then
      sel_B<="01";
      else sel_B<="10";
      end if;
      alu_operation<="0000";
      sel_cache_in<='1';
      sel_main<='1';
```

```
        end if;

        end case;

    end process;

end arc_control_unit;
```

--input unit

```
--debounce circuit

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


entity debounce is
port(clk,reset: in std_logic;
     tick: in std_logic;
     sw: in std_logic;
     db_sw: out std_logic;
     interrupt: out std_logic
       );
end debounce;

architecture arc_debounce of debounce is
type state
is(zero,wait1_1,wait1_2,wait1_3,one,wait0_1,wait0_2,wait0_3);
signal pr_state,nx_state: state;
begin


  process(clk,reset)
  begin
    if(reset='1')then
      pr_state<=zero;
    elsif(clk'event and clk='1')then
      pr_state<=nx_state;
    end if;
  end process;

  process(pr_state,sw,tick)
    begin
      case pr_state is
      when zero=>
        db_sw<='0';
        interrupt<='0';
        if(sw='1')then
          nx_state<=wait1_1;
        end if;

      when wait1_1=>
        if(sw='0')then
          nx_state<=zero;
        elsif(tick='1')then
          nx_state<=wait1_2;
        else nx_state<=wait1_1;
        end if;
```

```
      when wait1_2=>
        if(sw='0')then
          nx_state<=zero;
        elsif(tick='1')then
          nx_state<=wait1_3;
        else nx_state<=wait1_2;
        end if;

      when wait1_3=>
        if(sw='0')then
          nx_state<=zero;
        elsif(tick='1')then
          nx_state<=one;
          db_sw<='1';
          interrupt<='1';
        else nx_state<=wait1_3;
        end if;

        when one=>
        interrupt<='0';
        db_sw<='1';
        if(sw='0')then
          nx_state<=wait0_1;
        else nx_state<=one;
        end if;

      when wait0_1=>
        if(sw='1')then
          nx_state<=one;
        elsif(tick='1')then
          nx_state<=wait0_2;
        else nx_state<=wait0_1;
        end if;

      when wait0_2=>
        if(sw='1')then
          nx_state<=one;
        elsif(tick='1')then
          nx_state<=wait0_3;
        else nx_state<=wait0_2;
        end if;

      when wait0_3=>
        if(sw='1')then
          nx_state<=one;
        elsif(tick='1')then
          db_sw<='0';
          interrupt<='0';
          nx_state<=zero;
        else nx_state<=wait0_3;
        end if;
      end case;
    end process;

end arc_debounce;


--input unit component

library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity input_unit is
port(clk,reset: in std_logic;
     sw: in unsigned(4 downto 0);
     db_sw: out unsigned(31 downto 0);
     interrupt: out std_logic
     );
end input_unit;

architecture arc_input_unit of input_unit is
signal interrupt_0,interrupt_1,interrupt_2,interrupt_3,interrupt_4:
std_logic;
signal count: integer range 0 to 127;
signal tick: std_logic;
begin

  process(clk,reset)
  begin
    if(reset='1')then
      count<=0;
      tick<='0';
    elsif(clk'event and clk='1')then
      count<=count+1;
      if(count=99)then
        tick<='1';
        count<=0;
      else tick<='0';
      end if;
    end if;
  end process;

  debounce_0:entity work.debounce(arc_debounce) port
map(clk=>clk,reset=>reset,sw=>sw(0),db_sw=>db_sw(0),
             tick=>tick,interrupt=>interrupt_0);

  debounce_1:entity work.debounce(arc_debounce) port
map(clk=>clk,reset=>reset,sw=>sw(1),db_sw=>db_sw(1),
             tick=>tick,interrupt=>interrupt_1);

  debounce_2:entity work.debounce(arc_debounce) port
map(clk=>clk,reset=>reset,sw=>sw(2),db_sw=>db_sw(2),
             tick=>tick,interrupt=>interrupt_2);

  debounce_3:entity work.debounce(arc_debounce) port
map(clk=>clk,reset=>reset,sw=>sw(3),db_sw=>db_sw(3),
             tick=>tick,interrupt=>interrupt_3);

  debounce_4:entity work.debounce(arc_debounce) port
map(clk=>clk,reset=>reset,sw=>sw(4),db_sw=>db_sw(4),
             tick=>tick,interrupt=>interrupt_4);


  interrupt<=interrupt_0 or interrupt_1 or interrupt_2 or interrupt_3
or interrupt_4;

  db_sw(31 downto 5)<="00000000000000000000000000";

end arc_input_unit;
```

--sparc processor

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity sparc_processor is
port(clk: in std_logic;
     reset: in std_logic;
       switches: in unsigned(4 downto 0);
       data in: in unsigned(31 downto 0);
       write_en_mem: out std_logic;
       address: out unsigned(31 downto 0);
       data_out:out unsigned(31 downto 0);
       processor_out:out unsigned(7 downto 0)
       );
end sparc_processor;

architecture arc_sparc_processor of sparc_processor is
signal op: unsigned(1 downto 0);
signal op_2: unsigned(2 downto 0);
signal op_3: unsigned(5 downto 0);
signal cond: unsigned(3 downto 0);
signal i:  std_logic;
signal flags: unsigned(3 downto 0);
signal en_trap:  std_logic;
signal sel_pc:  unsigned(2 downto 0);
signal en_icc:  std_logic;
signal write_en_reg: std_logic;
signal write_en_st:  std_logic;
signal write_out:  std_logic;
signal sel_reg:  unsigned(1 downto 0);
signal sel_A:  std_logic;
signal sel_B:  unsigned(1 downto 0);
signal alu_operation:  unsigned(3 downto 0);
signal sel_main,sel_cache_in,cache_en,write_en_cache,hit,done:
std_logic;
signal trap_type:  unsigned(7 downto 0);
signal clk_2,en_pc:  std_logic;
signal db_switches,pc_intrr: unsigned(31 downto 0);
signal interrupt: std_logic;

begin

    clk_by_2: entity work.clk_2(arc_clk_2) port
map(clk=>clk,reset=>reset,clk_2=>clk_2);

    data_path: entity work.data_path(arc_data_path) port
map(clk=>clk_2,reset=>reset,en_pc=>en_pc,en_icc=>en_icc,

sel pc=>sel pc,write en reg=>write en reg,write en cache=>write en ca
che,write_en_st=>write_en_st,sel_reg=>sel_reg,

sel_A=>sel_A,data_in=>data_in,sel_B=>sel_B,alu_operation=>alu_operati
on,sel_cache_in=>sel_cache_in,sel_main=>sel_main,

trap_type=>trap_type,op=>op,op_2=>op_2,op_3=>op_3,cond=>cond,i=>i,fla
gs=>flags,en_trap=>en_trap,data_out=>data_out,
```

```
address=>address,write_out=>write_out,cache_en=>cache_en,hit=>hit,don
e=>done,write_en_mem=>write_en_mem,

processor_out=>processor_out,db_switches=>db_switches,pc_intrr=>pc_in
trr);

      control_unit: entity work.control_unit(arc_control_unit) port
map(clk=>clk_2,op=>op,op_2=>op_2,op_3=>op_3,cond=>cond,

i=>i,flags=>flags,en_trap=>en_trap,en_icc=>en_icc,en_pc=>en_pc,write_
en_reg=>write_en_reg,write_en_st=>write_en_st,

sel_pc=>sel_pc,sel_reg=>sel_reg,sel_A=>sel_A,sel_B=>sel_B,alu_operati
on=>alu_operation,sel_main=>sel_main,

sel_cache_in=>sel_cache_in,write_en_cache=>write_en_cache,cache_en=>c
ache_en,hit=>hit,done=>done,trap_type=>trap_type,
    write_out=>write_out,interrupt=>interrupt,pc_intrr=>pc_intrr);

    input_unit: entity work.input_unit(arc_input_unit) port
map(clk=>clk_2,reset=>reset,sw=>switches,db_sw=>db_switches,
               interrupt=>interrupt);


end arc_sparc_processor;
```

References

-MIT Press - Circuit Design with VHDL (2004)

-"Digital Design and Computer Architecture" by David Harris and Sarrah Harris.

-"The SPARC Architecture Manual" Version 8 by SPARC International Inc.