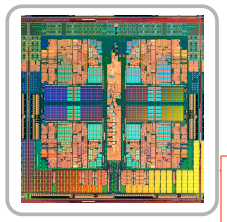
Lecture 10
The MIPS Datapath

Suggested Readings

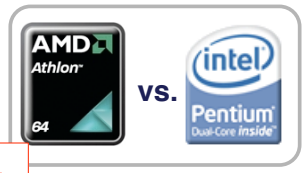
- **Readings**
  - **H&P: Chapter 4.1-4.4**

Processor components

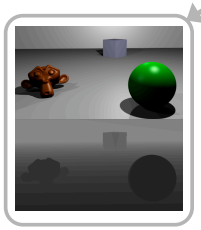
Multicore processors and programming



Processor comparison



Goal:
Describe the fundamental components required in a single core of a modern microprocessor as well as how they interact with each other, with main memory, and with external storage media.



Writing more efficient code

The right HW for the right application

```
for i=0; i<5; i++ {
      a = (a*b) + c;
}
```

```
MULT r1,r2,r3   # r1 ← r2*r3
ADD r2,r1,r4    # r2 ← r1+r4
```

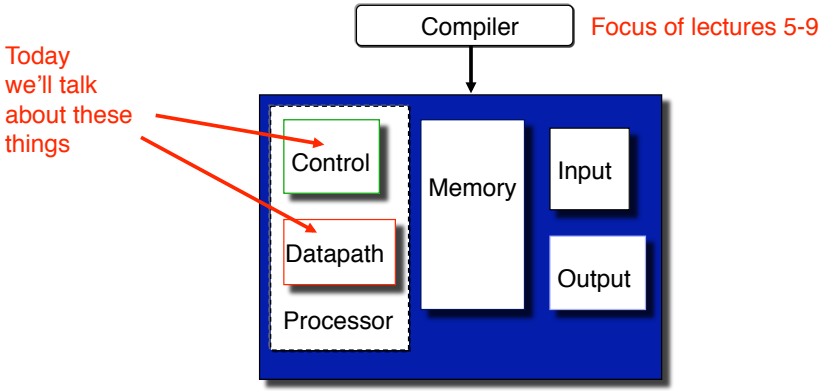
110011	000001	000010	000011
001110	000010	000001	000100

HLL code translation

The organization of a computer

Von Neumann Model:

- Stored-program machine instructions are represented as numbers
- Programs can be stored in memory to be read/written just like numbers.



Review: Functions of Each Component

- Datapath: performs data manipulation operations
  - arithmetic logic unit (ALU)
  - floating point unit (FPU)
- Control: directs operation of other components
  - finite state machines
  - micro-programming
- Memory: stores instructions and data
  - random access v.s. sequential access
  - volatile v.s. non-volatile
  - RAMs (SRAM, DRAM), ROMs (PROM, EEPROM), disk
  - tradeoff between speed and cost/bit
- Input/Output and I/O devices: interface to environment
  - mouse, keyboard, display, device drivers

The MIPS Subset

- To simplify things a bit we'll just look at a few instructions:
  - memory-reference: lw, sw
  - arithmetic-logical: add, sub, and, or, slt
  - branching: beq, j

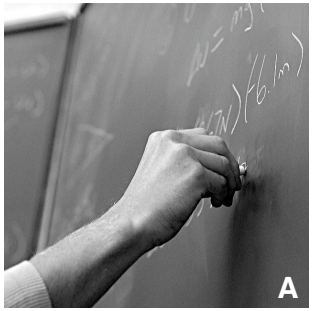
Most common instructions

- Organizational overview:
  - fetch an instruction based on the content of PC
  - decode the instruction
  - fetch operands
    - (read one or two registers)
  - execute
    - (effective address calculation/arithmetic-logical operations/ comparison)
  - store result
    - (write to memory / write to register / update PC)

With Von Neumann, RISC model do similar things for each instruction

Board discussion:

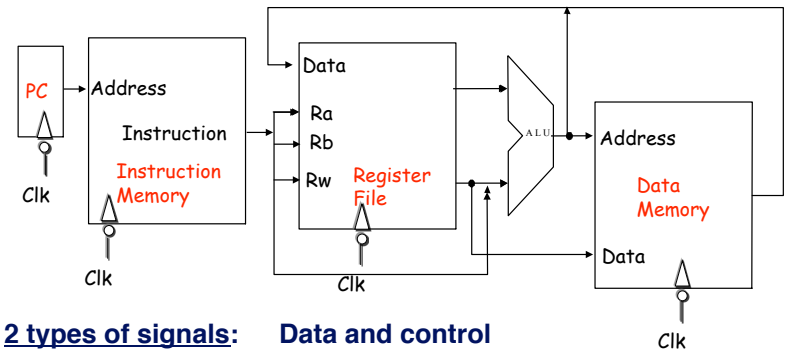
- Let's derive the MIPS datapath…



Implementation Overview

- Abstract / Simplified View:

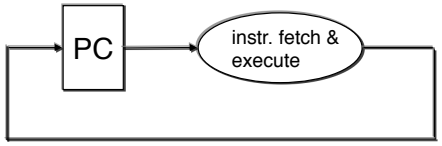
simplest view of Von Neumann, RISC μP



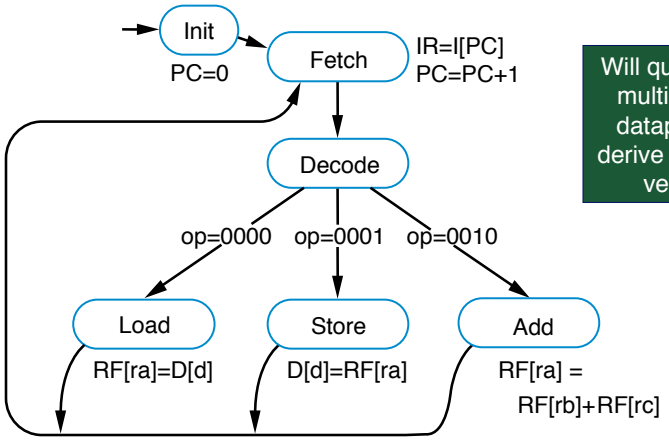
- 2 types of signals: Data and control
- Clocking strategy: Derived datapath is single cycle; did not talk about internal storage

Single Cycle Implementation

- Each instruction takes one cycle to complete.
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce "right answer" right away
- Cycle time determined by length of the longest path



But before, datapath was "multi-cycle"….

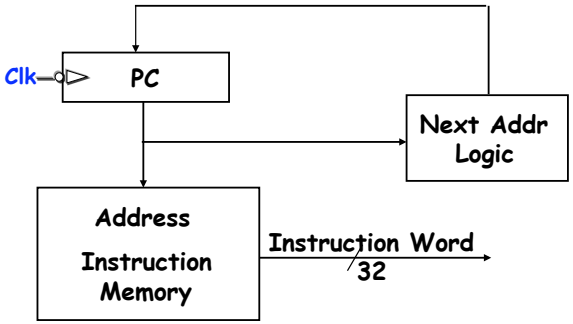


Will quickly move to multi-cycle MIPS datapath, but will derive a single-cycle version 1st…

Review:
Derivation of Single Cycle Datapath

Instruction Fetch Unit

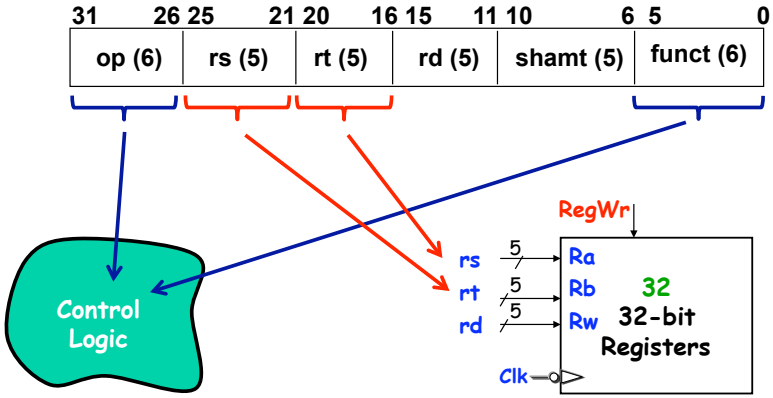
- Fetch the instruction: mem[PC] ,
- Update the program counter:
  - sequential code: PC <- PC+4
  - branch and jump: PC <- "something else"



During Decode…

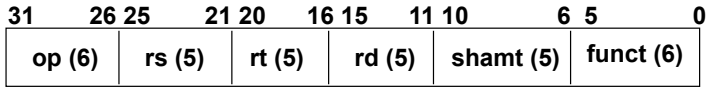
- Take bits from instruction encoding in IR and send to different parts of datapath

e.g. R-type, Add encoding:



Let's say we want to fetch…
…an R-type instruction (arithmetic)

- Instruction format:

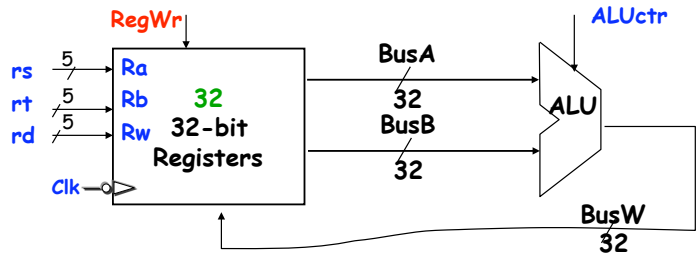


- RTL:
  - Instruction fetch: mem[PC]
  - ALU operation: reg[rd] <- reg[rs] op reg[rt]
  - Go to next instruction: Pc <- PC+ 4

So IR ← Memory(PC)

- Ra, Rb and Rw are from instruction's rs, rt, rd fields → sort of like passing args into a function.
- Actual ALU operation and register write should occur after decoding the instruction.

Datapath for R-Type Instructions



- Register timing:
  - Register can always be read.
  - Register write only happens when RegWr is set to high and at the falling edge of the clock
    - What does this say about CC time?

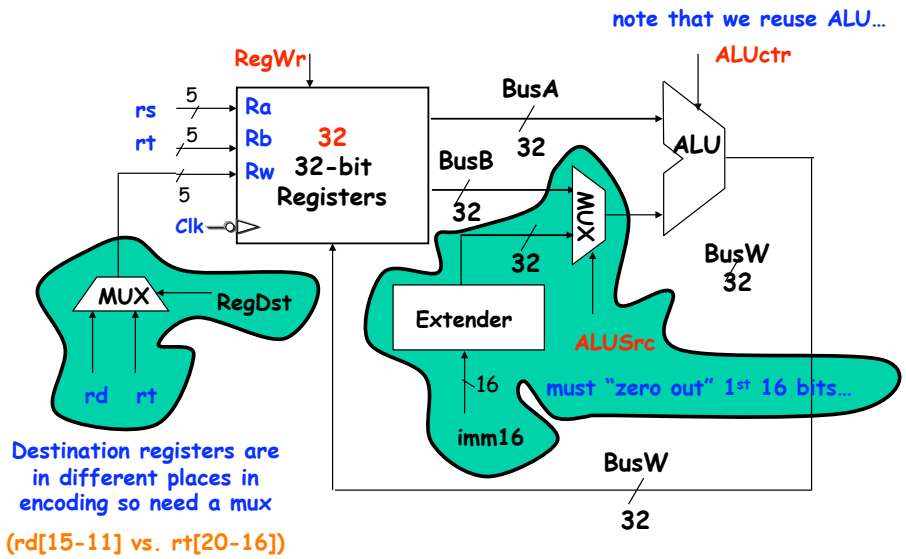
I-Type Arithmetic/Logic Instructions

- Instruction format: (Just I-type *Arithmetic* Instructions)

31 – 26	25 – 21	20 – 16	15 – 0
Op (6)	rs (5)	rt (5)	Address/Immediate value (16)

- RTL for arithmetic operations: e.g., ADDI
  - Instruction fetch: mem[PC]
  - Add operation: reg[rt] <- reg[rs] + SignExt(imm16)
  - Go to next instruction: Pc <- PC+ 4

Datapath for I-Type A/L Instructions



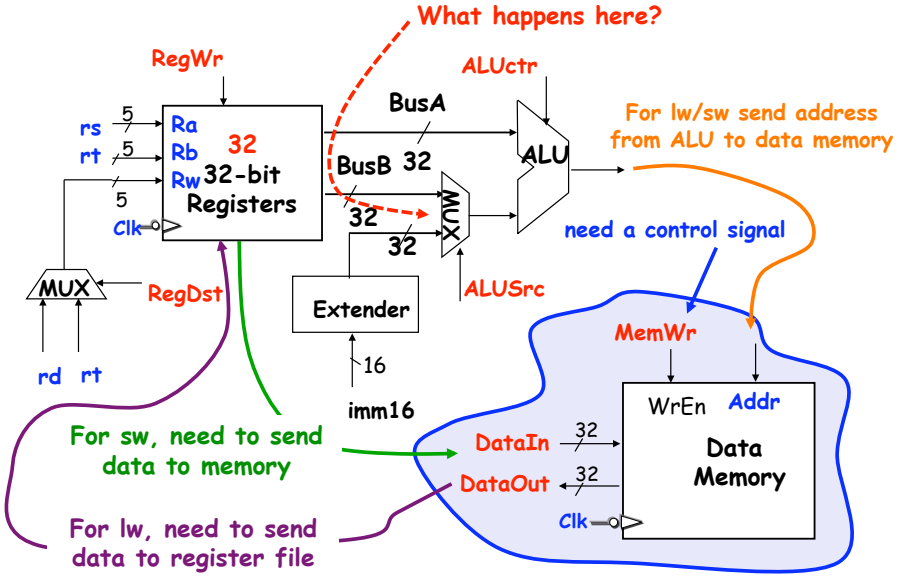
I-Type Load/Store Instructions

- Instruction format: (Just I-type *Arithmetic* Instructions)

31 – 26	25 – 21	20 – 16	15 – 0
Op (6)	rs (5)	rt (5)	Address/Immediate value (16)

- RTL for load/store operations: e.g., LW
  - Instruction fetch: mem[PC]
  - Compute memory address: Addr <- reg[rs] + SignExt(imm16)
  - Load data into register: reg[rt] <- mem[Addr]
  - Go to next instruction: Pc <- PC+ 4
- How about store? { same thing, just make 3rd step mem[addr] ← reg[rt]

Datapath for Load/Store Instructions



I-Type Branch Instructions

- Instruction format:

31 26	25 21	20 16	15 0
Op (6)	rs (5)	rt (5)	Address/Immediate value (16)

- RTL for branch operations: e.g., BEQ
  - Instruction fetch: mem[PC]
  - Compute conditon: Cond <- reg[rs] - reg[rt]
  - Calculate the next instruction's address:

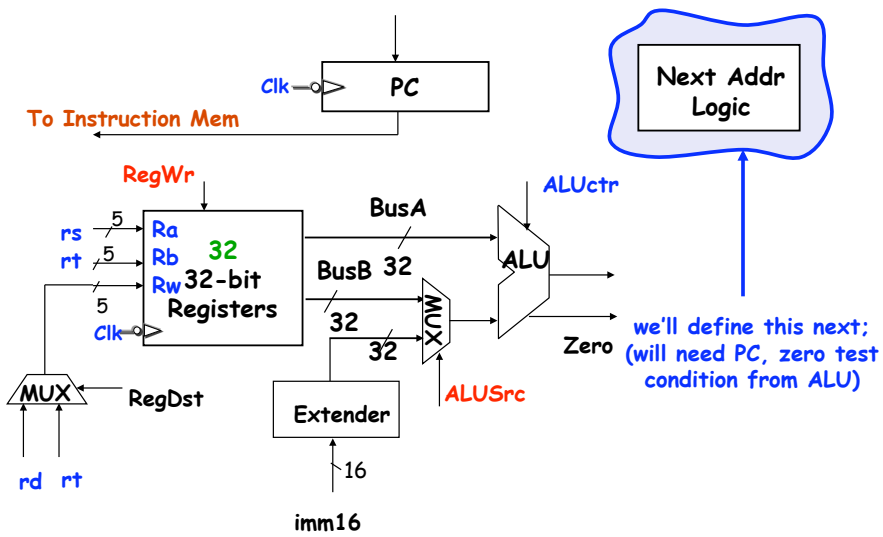
    if (Cond eq 0) then

    PC <- PC+ 4 + (SignExd(imm16) x 4)

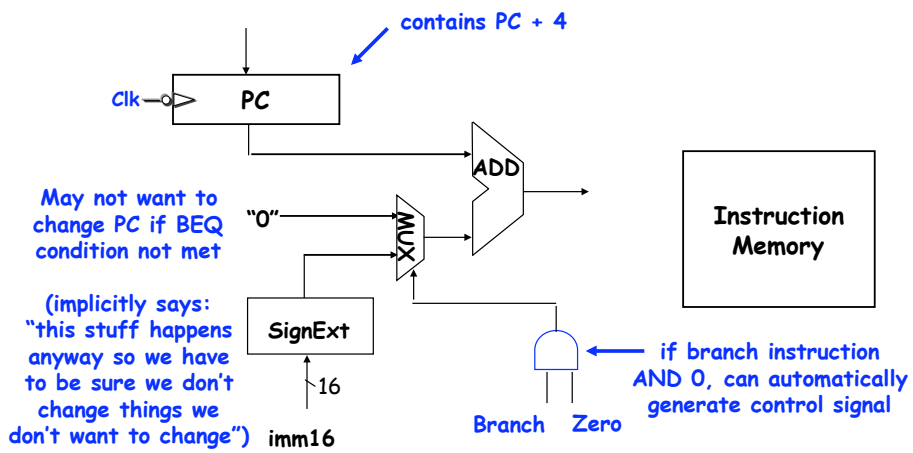
    else ?

need to align

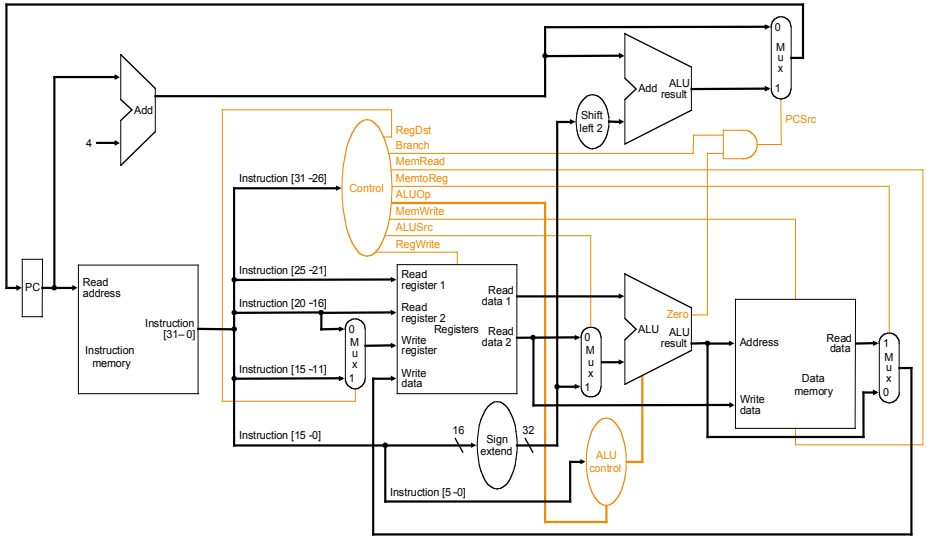
Datapath for Branch Instructions



Next Address Logic



A Single Cycle Datapath



Let's trace a few instructions:

- For example…
  - Add $5, $6, $7
  - SW 0($9), $10
  - Sub $1, $2, $3
  - LW $11, 0($12)

Single cycle versus multi-cycle

Single-Cycle Implementation

- Single-cycle, fixed-length clock:
  - CPI = 1
  - Clock cycle = propagation delay of the longest datapath operations among all instruction types
  - Easy to implement
- How to determine cycle length?
- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)

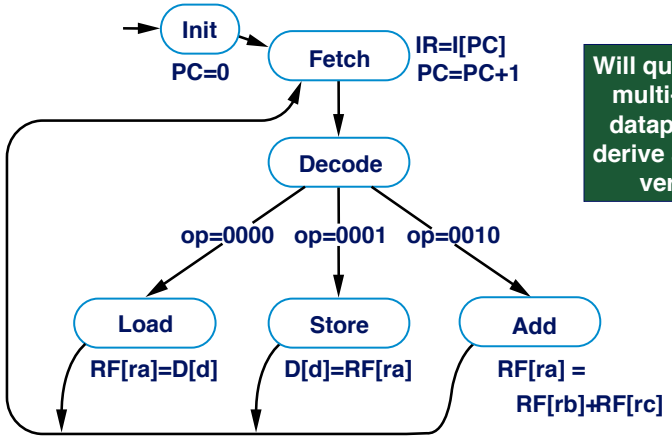
  - R-type: max {mem + RF + ALU + RF, Add} = 6ns
  - LW: max{mem + RF + ALU + mem + RF, Add} = 8ns
  - SW: max{mem + RF + ALU + mem, Add} = 7ns
  - BEQ: max{mem + RF + ALU, max{Add, mem + Add}} = 5ns

What is the CC time?

Multiple Cycle Alternative

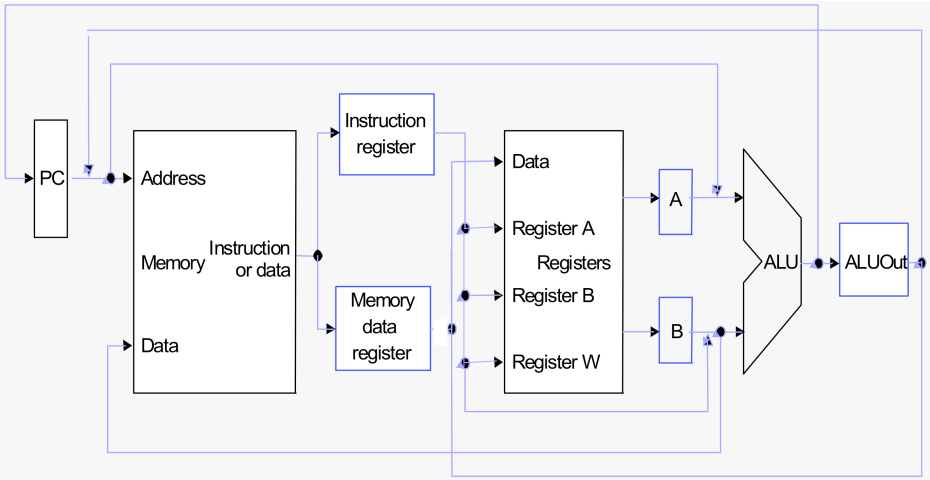
- Break an instruction into smaller steps
- Execute each step in one cycle.
- Execution sequence:
  - Balance amount of work to be done
  - Restrict each cycle to use only one major functional unit
  - At the end of a cycle
    - Store values for use in later cycles
    - Introduce additional "internal" registers
- The advantages:
  - Cycle time much shorter
  - Diff. inst. take different # of cycles to complete
  - Functional unit used more than once per instruction

But before, datapath was "multi-cycle"….



Will quickly move to multi-cycle MIPS datapath, but will derive a single-cycle version 1st…

A Multiple Cycle MIPS Datapath



Five Step Execution

1. Instruction Fetch (Ifetch):
   - Fetch instruction at address ($PC)
   - Store the instruction in register IR
   - Increment PC
2. Instruction Decode and Register Read (Decode):
   - Decode the instruction type and read register
   - Store the register contents in registers A and B
   - Compute new PC address and store it in ALUOut
3. Execution, Memory Address Computation, or Branch Completion (Execute):
   - Compute memory address (for LW and SW), or
   - Perform R-type operation (for R-type instruction), or
   - Update PC (for Branch and Jump)
   - Store memory address or register operation result in ALUOut

Five Step Execution (cont'd)

4. Memory Access or R-type instruction completion (MemRead/RegWrite/MemWrite):
   - Read memory at address ALUOut and store it in MDR
   - Write ALUOut content into register file, or
   - Write memory at address ALUOut with the value in B
5. Write-back step (WrBack):
   - Write the memory content read into register file

- Number of cycles for an instruction:
  - R-type: 4
  - lw: 5
  - sw: 4
  - Branch or Jump: 3

Execution Sequence Summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Mem[PC], PC = PC + 4			
Instruction decode/register fetch	A =RF [IR[25:21]], B = RF [IR[20:16]], ALUOut = PC + (sign-extend (IR[1:-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15:0])	if (A =B) then PC = ALUOut	PC = PC [31:28] | (IR[25:0]<<2)
Memory access or R-type completion	RF [IR[15:11]] = ALUOut	Load: MDR = Mem[ALUOut] or Store: Mem[ALUOut]= B		
Memory read completion		Load: RF[IR[20:16]] = MDR		

Some Simple Questions

- How many cycles will it take to execute this code?

```
lw  $t2, 0($t3)
lw  $t3, 4($t3)
beq $t2, $t3, Label #assume branch is not taken
add $t5, $t2, $t3
sw $t5, 8($t3)
Label:  ...
```

5+5+3+4+4=21

- What is being done during the 8th cycle of execution?

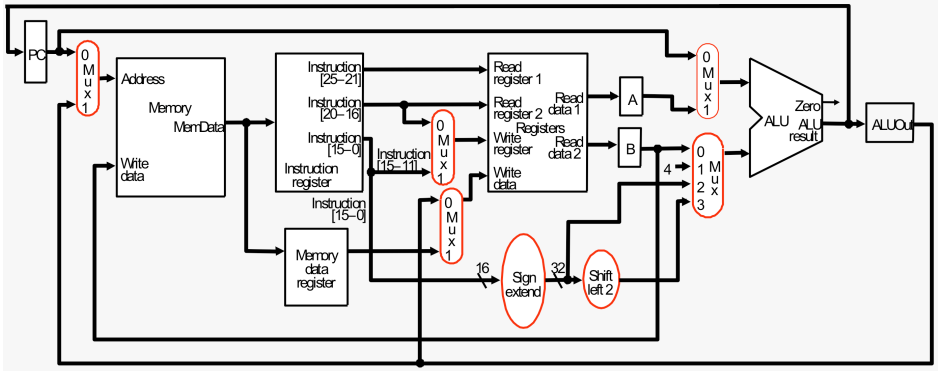
  Compute memory address: 4+$t3

- In what cycle does the actual addition of $t2 and $t3 takes place? 16

What if multi-cycle clock period is 2 ns vs. 8 ns for a single cycle?

Multiple Cycle Design

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional "internal" registers

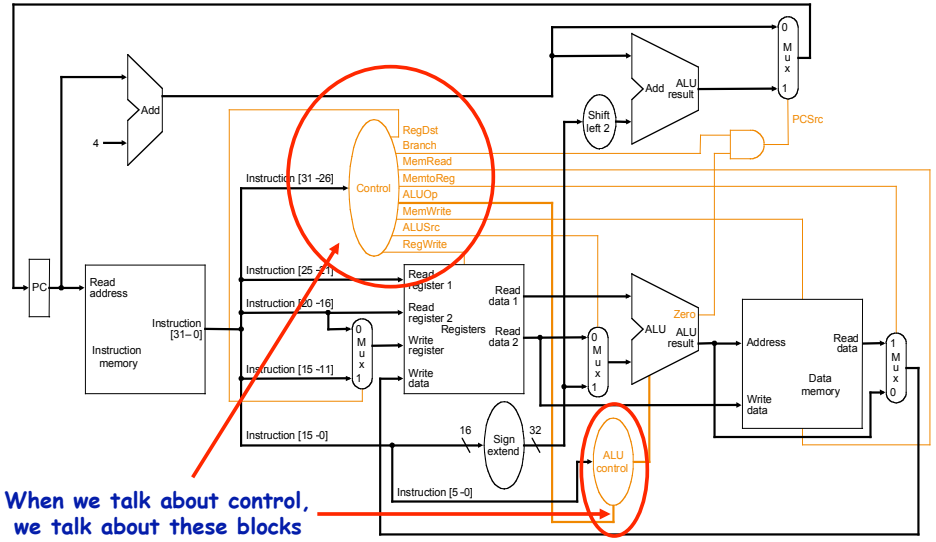


Control Logic

(I.e. *now*, we need to make the HW do what we want it to do - add, subtract, etc. - when we want it to…)

The HW needed, plus control

Single cycle MIPS machine



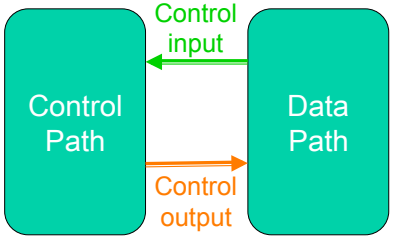
When we talk about control, we talk about these blocks

Implementing the Control (Part 1)

- Implementation Steps:
  - Identify control inputs and control output (control words)
  - Make a control signal table for each cycle
  - Derive control logic from the control table
- Do we need a FSM here?

This logic can take on many forms: combinational logic, ROMs, microcode, or combinations…

Control inputs:
Opcode (5 bits)
Func (6 bits)



Control outputs:
RegDst
MemtoReg
RegWrite
MemRead
MemWrite
ALUSrc
ALUctr
Branch
Jump

Implementing Control

- Implementation Steps:
  1. Identify control inputs and control outputs
  2. Make a control signal table for each cycle
  3. Derive control logic from the control table
     - This logic can take on many forms: combinational logic, ROMs, microcode, or combinations…

Single Cycle Control Input/Output

- Control Inputs:
  - Opcode (6 bits)
  - How about R-type instructions?
- Control Outputs:
  - RegDst
  - ALUSrc
  - MemtoReg
  - RegWrite
  - MemRead
  - MemWrite
  - Branch
  - Jump
  - ALUctr

Step 1: Idenitfy inputs & outputs

these are columns

these are rows

Step 2: Make a control signal table for each cycle

Control Signal Table

R-type

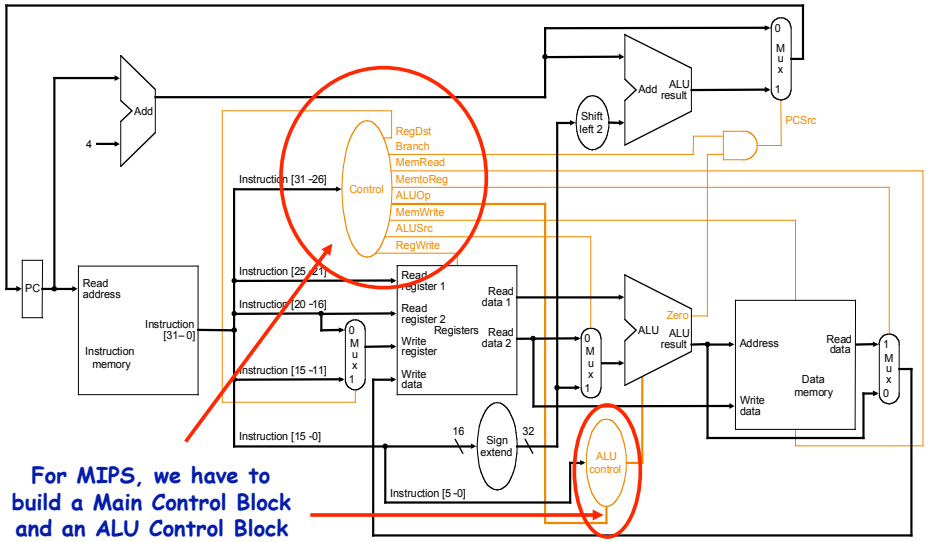
(inputs)

	Add	Sub	LW	SW	BEQ
Func (input)	100000	100010	xxxxxx	xxxxxx	xxxxxx
Op (input)	000000	000000	100011	101011	000100
RegDst	1	1	0	X	X
ALUSrc	0	0	1	1	0
Mem-to-Reg	0	0	1	X	X
Reg. Write	1	1	1	0	0
Mem. Read	0	0	1	0	0
Mem. Write	0	0	0	1	0
Branch	0	0	0	0	1
ALUOp	Add	Sub	00	00	01

(outputs)

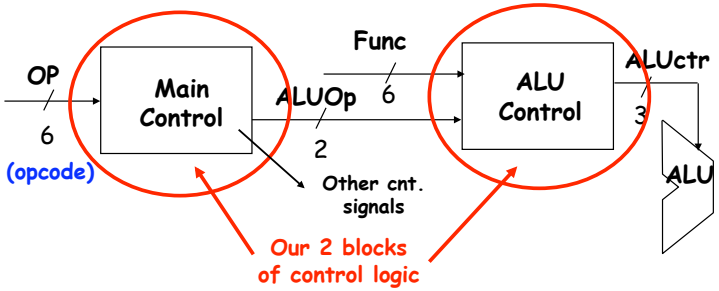
The HW needed, plus control

Single cycle MIPS machine



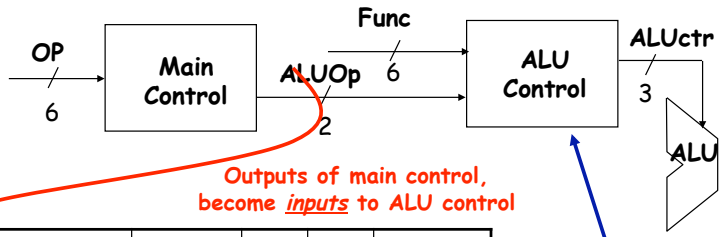
For MIPS, we have to build a Main Control Block and an ALU Control Block

Main control, ALU control



- Use OP field to generate ALUOp (encoding)
  - Control signal fed to ALU control block
- Use Func field and ALUOp to generate ALUctr (decoding)
  - Specifically sets 3 ALU control signals
    - B-Invert, Carry-in, operation

Main control, ALU control



Outputs of main control, become *inputs* to ALU control

	R-type	lw	sw	beq
ALU Operation	"R-type"	add	add	subtract
ALUOp<1:0>	10	00	00	01

We have 8 bits of input to our ALU control block; we need 3 bits of output…

Or in other words…
00 = ALU performs add
01 = ALU performs sub
10 = ALU does what function code says

Generating ALUctr

- We want these outputs:

ALU Operation	and	or	add	sub	slt
ALUctr<2:0>	000	001	010	110	111

ALUctr<2> = B-negate (C-in & B-invert)
ALUctr<1> = Select ALU Output
ALUctr<0> = Select ALU Output

Invert B and C-in must be a 1 for subtract

and - 00
or - 01
adder - 10
less - 11
mux

- We have these inputs…

func<5:0>

36 (and) = 1 0 0 1 0 0
37 (or) = 1 0 0 1 0 1
32 (add) = 1 0 0 0 0 0
34 (sub) = 1 0 0 0 1 0
42 (slt) = 1 0 1 0 1 0

can ignore these (they're the same for all…)

Inputs — Outputs

	ALUOp		Funct field						ALUctr
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
lw/sw	0	0	X	X	X	X	X	X	010 (add)
beq	0	1	X	X	X	X	X	X	110 (sub)
R-type	1	X	X	X	0	0	0	0	010 (add)
	1	X	X	X	0	0	1	0	110 (sub)
	1	X	X	X	0	1	0	0	000 (and)
	1	X	X	X	0	1	0	1	001 (or)
	1	X	X	X	1	0	1	0	111 (slt)

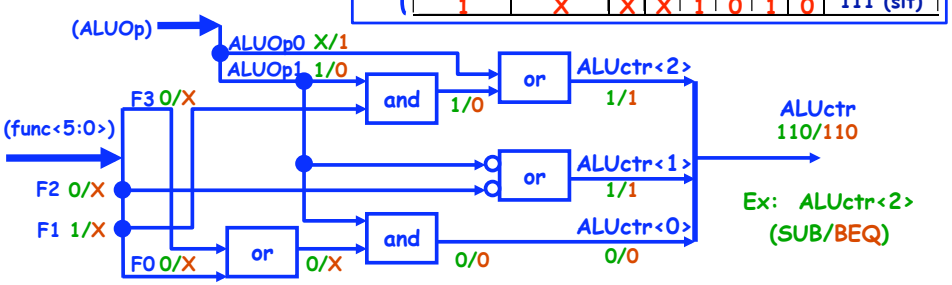
The Logic

This table is used to generate the actual Boolean logic gates that produce ALUctr.

Could generate gates by hand, often done w/SW.

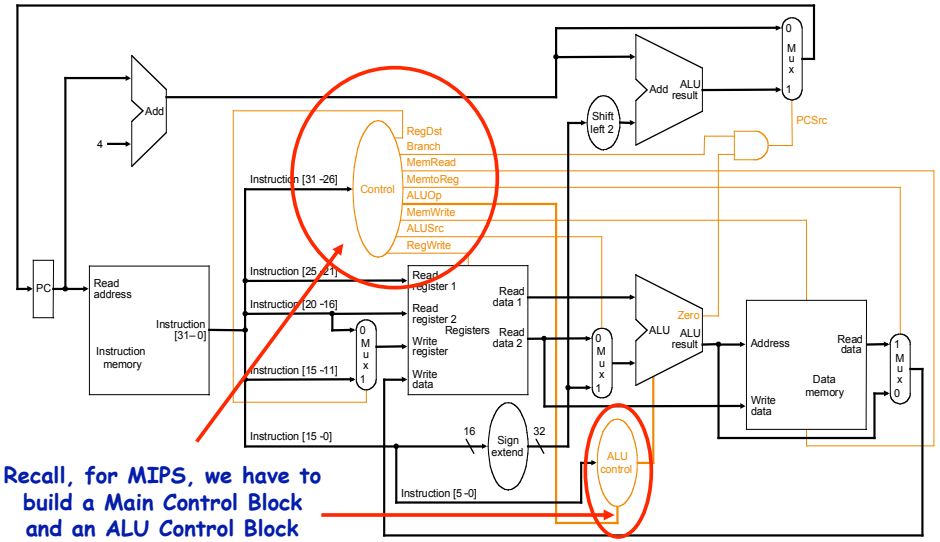
Inputs — Outputs

	ALUOp		Funct field						ALUctr
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
lw/sw	0	0	X	X	X	X	X	X	010 (add)
beq	0	1	X	X	X	X	X	X	110 (sub)
R-type	1	X	X	X	0	0	0	0	010 (add)
	1	X	X	X	0	0	1	0	110 (sub)
	1	X	X	X	0	1	0	0	000 (and)
	1	X	X	X	0	1	0	1	001 (or)
	1	X	X	X	1	0	1	0	111 (slt)



Recall…

Single cycle MIPS machine

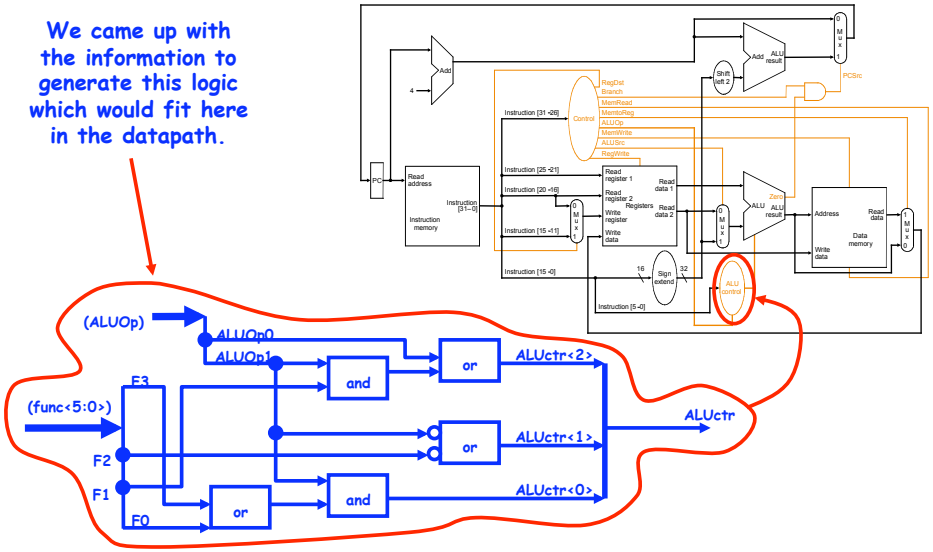


Recall, for MIPS, we have to build a Main Control Block and an ALU Control Block

Well, here's what *we* did…

Single cycle MIPS machine

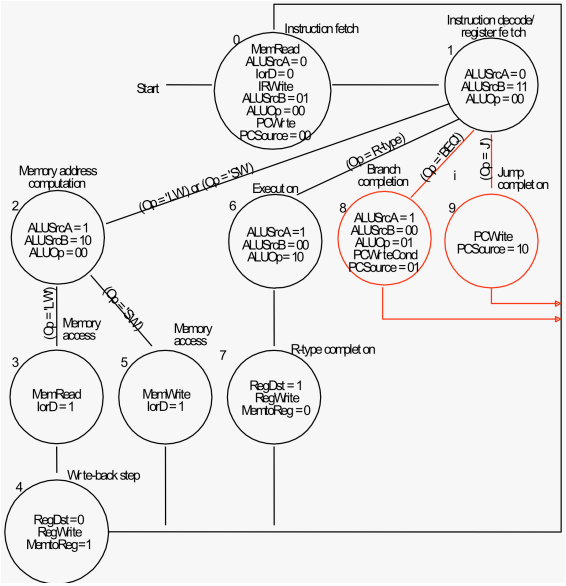
We came up with the information to generate this logic which would fit here in the datapath.



Implementing the Control (Part 2)

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- How to represent all the information?
  - finite state diagram
  - microprogramming
- Realization of a control unit is independent of the representation used
  - Control outputs: random logic, ROM, PLA
  - Next-state function: same as above or an explicit sequencer

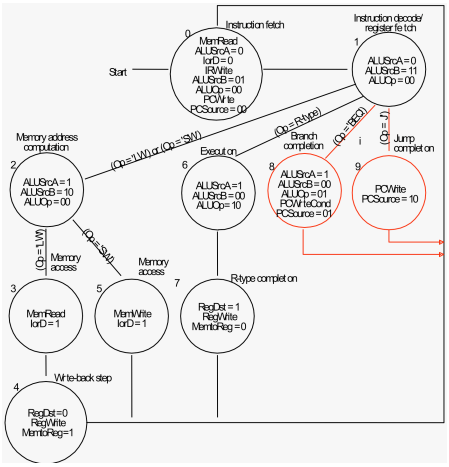
Finite State Diagram

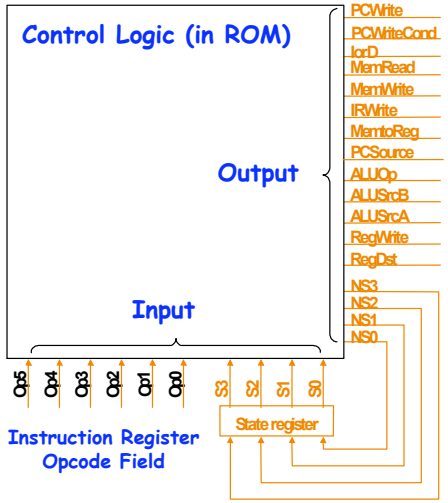


Microprogramming as an Alternative

- Control unit can easily reach thousands of states with hundreds of different sequences.
  - A large set of instructions and/or instruction classes (x86)
  - Different implementations with different cycles per instruction
- Flexibility may be needed in the early design phase
- An alternative: Microcode.
  - Treat the set of control signals to be asserted in a state as an *instruction* to be executed (referred to as microinstructions)
  - Treat state transitions as an instruction sequence

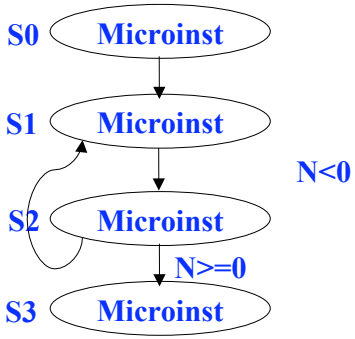
Foreshadowing: The Net Result





Microprogramming as an Alternative (cont'd)

- Each state → one microinstruction
- State transitions → microinstruction sequencing
- Setting up control signals → executing microinstructions
- To specify control, we just need to write microprograms (or microcode)



Microinstruction Format (1)

- Group the control signals according to how they are used
- For the 5-cycle MIPS organization:
  - Memory: IorD, MemRead, MemWrite
  - Instruction Register: IRWrite
  - PC: PCWrite, PCWriteCond, PCSource
  - Register File: RegWrite, MemtoReg, RegDst
  - ALU: ALUSrcA, ALUSrcB, ALUOp
- Group them as follows:
  - Memory (for both Memory and Instruction Register)
  - PC write control (for PC)
  - Register control (for Register File)
  - ALU control
  - SRC1
  - SRC2
  - Sequencing

  } For ALU Control

Microinstruction Format (2)

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Sub	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's func to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB= 00	Register B is the second ALU input.
	4	ALUSrc = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB= 10	Use output of the sign ext unit as the 2nd ALU input.
	Extshft	ALUSrcB= 11	Use output of shift-by-two unit as the 2nd ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg=0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg=1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.

Microinstruction Format (3)

Field name	Value	Signals active	Comment
Memory	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource=01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource=10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Sample Microinstruction (1)

IFetch: IR = Mem[PC], PC = PC+4

PCWrite: 1
PCWriteCond:
lorD: 0
MemRead: 1
IRWrite: 1
MemtoReg:
PCSource:
ALUOp: 00
ALUSrcB: 01
ALUSrcA: 0
RegWrite:
RegDst:
AddrCtrl: 11

Microinstruction:

ALUctrl	SRC1	SRC2	RegCtrl	Memory	PCWrite	Sequen
Add	PC	4	--	ReadPC	ALU	Seq

Sample Microinstruction (2)

Decode: A= RF[IR[25:21]], B= RF[IR[20:16]],
ALUOut = PC + Sign_Ext(IR[15:0]) << 2);

PCWrite:
PCWriteCond:
lorD:
MemRead:
IRWrite:
MemtoReg:
PCSource:
ALUOp: 00
ALUSrcB: 11
ALUSrcA: 0
RegWrite:
RegDst:
AddrCtrl: 01

Microinstruction:

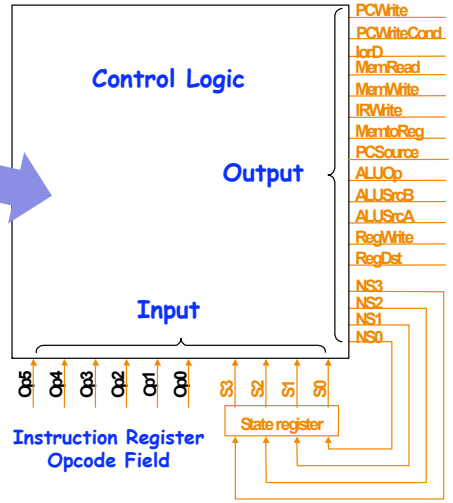
ALUctrl	SRC1	SRC2	RegCtrl	Memory	PCWrite	Sequen
Add	PC	ExtShf	Read			Disp 1

Put It All Together

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Sub	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Control Implementations

Could use a programmable ROM



Exceptions

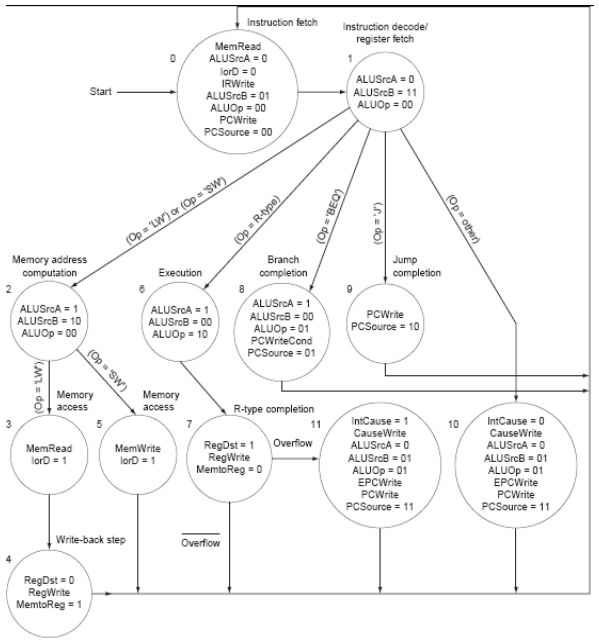
- Exceptions: unexpected events from within the processor
  - arithmetic overflow
  - undefined instruction
  - switching from user program to OS
- Interrupts: unexpected events from outside of the processor
  - I/O request
- Consequence: alter the normal flow of instruction execution
- Key issues:
  - detection
  - action
    - save the address of the offending instruction in the EPC
    - transfer control to OS at some specified address
- Exception type indication:
  - status register
  - interrupt vector

Another reason that register naming conventions are important.

Exception Handling

- Types of exceptions considered:
  - undefined instruction
  - arithmetic overflow
- MIPS implementation:
  - EPC: 32-bit register, EPCWrite
  - Cause register: 32-bit register, CauseWrite
    - undefined instruction: Cause register = 0
    - arithmetic overflow: Cause register = 1
  - IntCause: 1 bit control
  - Exception Address: C0000000 (hex)
- Detection:
  - undefined instruction: op value with no next state
  - arithmetic overflow: overflow from ALU
- Action:
  - set EPC and Cause register
  - set PC to Exception Address

FSM with Exception Handling



Datapath with Exception Handling