# L07-L09 recap:  Fundamental lesson(s)

- Over the next 3 lectures (using the MIPS ISA as context) I'll explain:
  - How functions are treated and processed in assembly
  - How system calls are enabled in assembly
  - How exceptions are handled in assembly

- I'll also explain why it's important that register conventions be followed
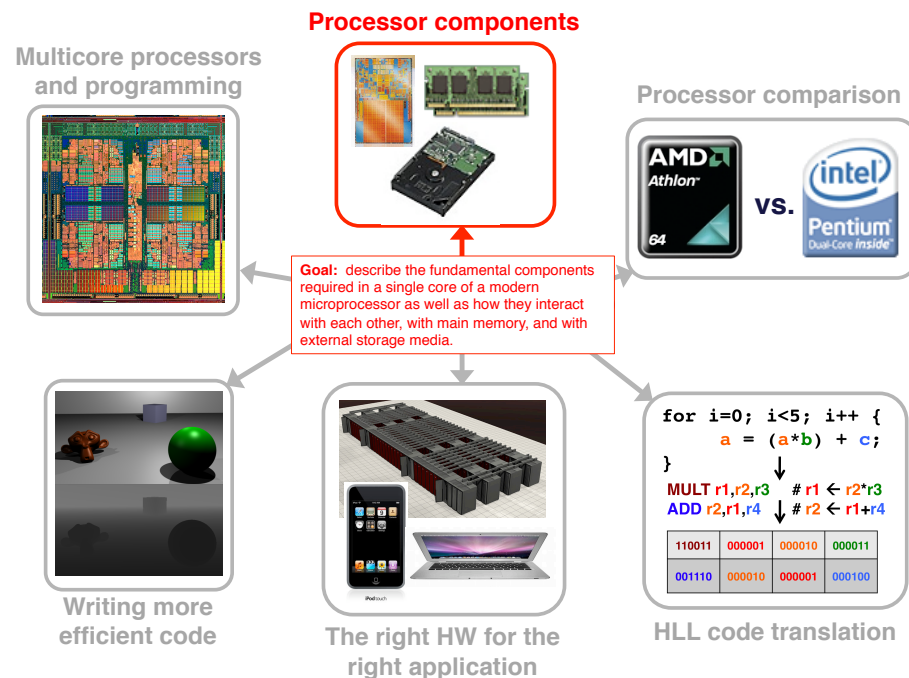
1

# L07-L09 recap:  Why it's important…

- If you ever write a compiler or OS some day, you will need to be aware of, and code for all of the issues to be discussed over the next 3 lectures

- If you understand what architectural overhead may be associated with (compiled) procedure calls, you should be able to write much more efficient HLL code

2

# Lecture 10
# The MIPS datapath

**Suggested reading:**
**(HP Chapter 4.1—4.4)**

3

**Processor components**

Multicore processors and programming

Processor comparison

AMD Athlon 64  vs.  intel Pentium Dual-Core inside

**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

4

# Fundamental lesson(s)

- Today we'll discuss what hardware is required to execute an instruction, as well as how to best "organize" it.

# Why it's important…

- If you ever design the HW for a microprocessor, etc. you'll need to be aware of these types of issues

- Understanding organization – and how it impacts the delay of something like a memory reference - will make you a better programmer

- It is now more and more important to design HW/SW simultaneously

# 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.



Compiler — Focus of lectures 5-9

Today we'll talk about these things

Control
Datapath
Processor
Memory
Input
Output
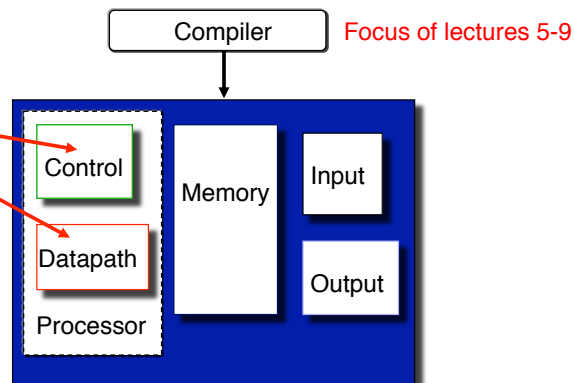
# Review:  Functions of Each Component

- **Datapath: performs data manipulation operations**
  - **arithmetic logic unit (ALU)**
  - **floating point unit (FPU)**

- **Control: directs operation of other components**
  - **finite state machines**
  - **micro-programming (to be discussed)**

- **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**

# Let's "derive" the MIPS datapath:

- 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**

  Very 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

A

9

# A Single Cycle Datapath
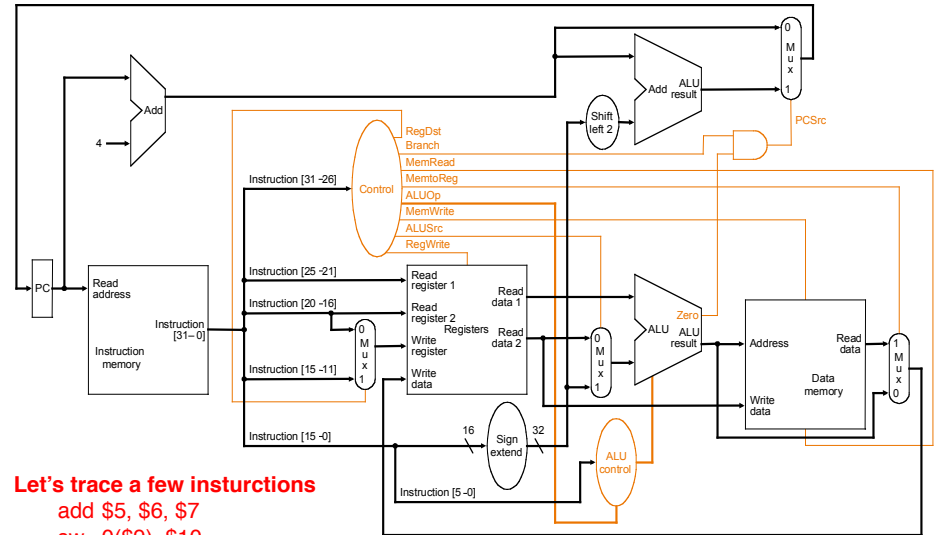


Let's trace a few insturctions
  add $5, $6, $7
  sw   0($9), $10
  sub  $1, $2, $3
  lw   $11, 0($12)

10

# PERFORMANCE

11

# 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?**

12

# Before, spoke about "multi-cycle" datapath



- Init
- PC=0
- Fetch — IR=I[PC], PC=PC+1
- Decode
- op=0000, op=0001, op=0010
- Load — RF[ra]=D[d]
- Store — D[d]=RF[ra]
- Add — RF[ra] = RF[rb]+RF[rc]
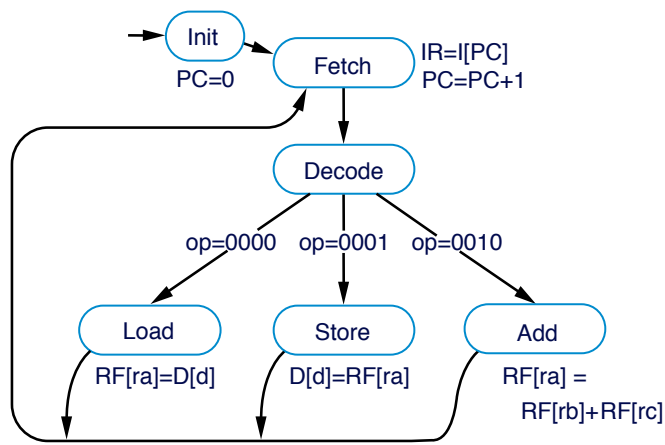
# The multi-cycle approach (& benefits):

- **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**

# MIPS multi-cycle datapath
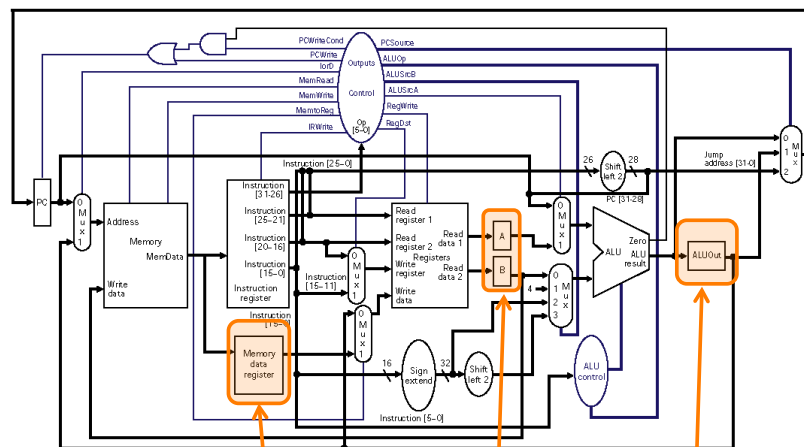


Note introduction of temporary storage registers

# 5 steps an instruction could take:

**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**

# 5 steps an instruction could take:

**4. Memory Access or R-type instruction completion**
**(MemRead/RegWrite/MemWrite):**
– **Read memory at address ALUOut and store it in <u>MDR</u>**
– **Write ALUOut content into register file, or**
– **Write memory at address ALUOut  with the value in <u>B</u>**

**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**

# Instruction execution (summary):

| Step name | Action for R-type instructions | Action for memory-reference instructions | Action for branches | Action for jumps |
|---|---|---|---|---|
| Instruction fetch | | IR = Mem[PC], <br> PC = PC + 4 | | |
| Instruction decode/register fetch | | A =RF [IR[25:21]], <br> B = RF [IR[20:16]], <br> 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] <br> or <br> Store: Mem[ALUOut]= B | | |
| Memory read completion | | Load: RF[IR[20:16]] = MDR | | |

# Some 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 + content of $t3**
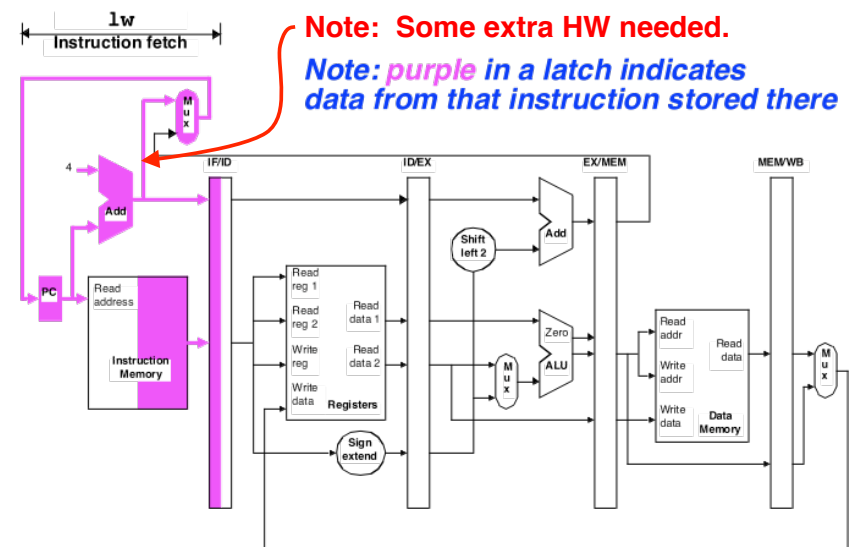
- **In what cycle does the addition of $t2 and $t3 takes place?**
  **16**

- **How would performance compare if multi-cycle clock period is 2 ns and cycle cycle period is 10 ns?**

  **(21 CCs x 2 ns) vs. (5 CCs x 10 ns):     42 ns vs. 50 ns**

# Foreshadowing:  pipelines



**Note:  Some extra HW needed.**
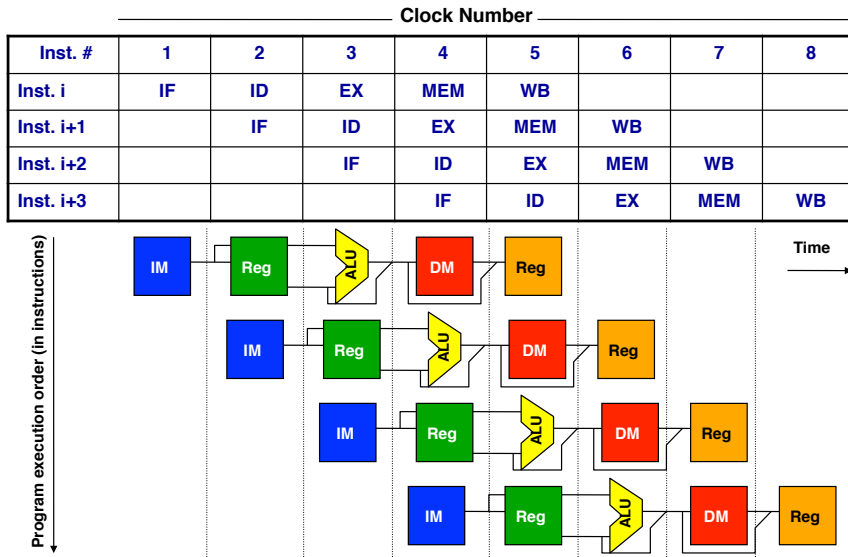*Note: purple in a latch indicates data from that instruction stored there*

Data must be stored from one stage to the next in pipeline registers/latches.
hold temporary values between clocks and needed info. for  execution.

## Another way to look at it…

| Inst. # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---------|-----|-----|-----|-----|-----|-----|-----|-----|
| Inst. i | IF | ID | EX | MEM | WB | | | |
| Inst. i+1 | | IF | ID | EX | MEM | WB | | |
| Inst. i+2 | | | IF | ID | EX | MEM | WB | |
| Inst. i+3 | | | | IF | ID | EX | MEM | WB |



Program execution order (in instructions)

Time

21

---

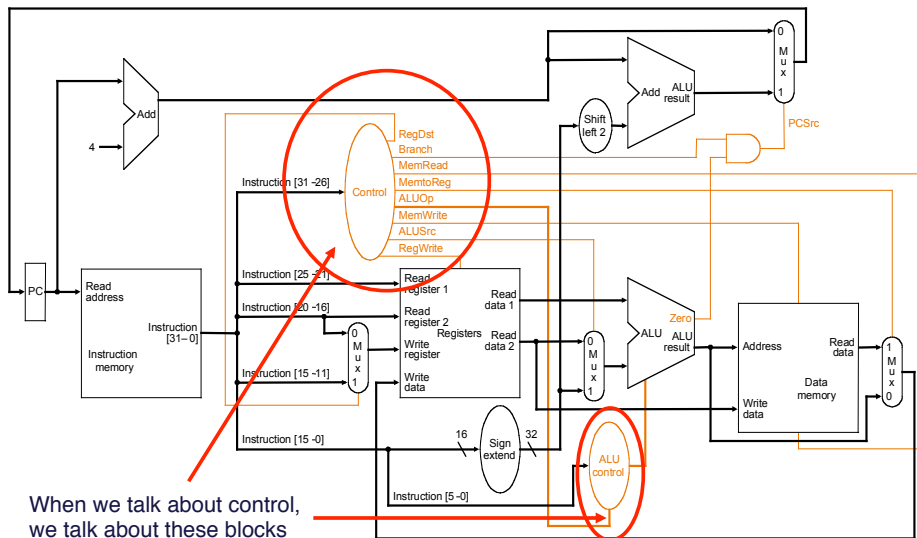(Tying computer architecture to logic design…)

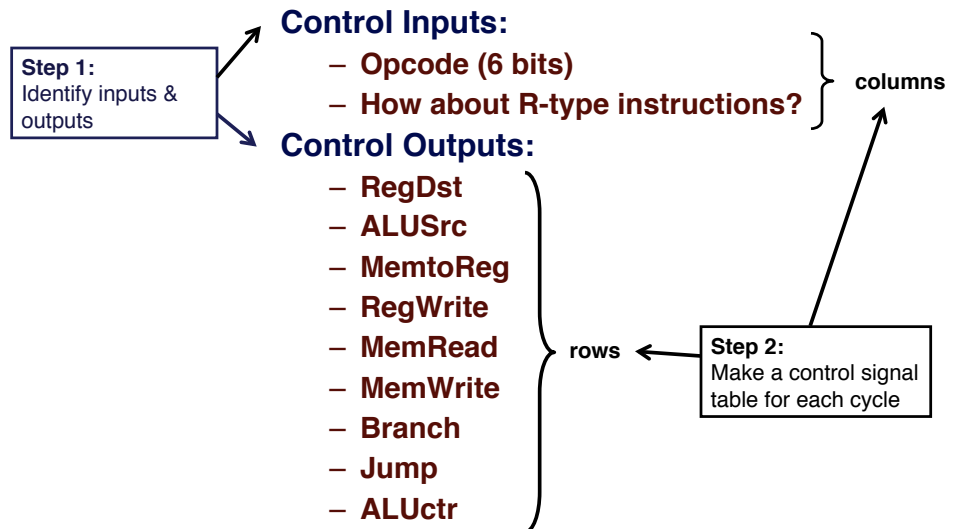# A SHORT DISCUSSION ABOUT CONTROL LOGIC

22

---

## The HW needed, plus control

Single cycle MIPS machine



When we talk about control, we talk about these blocks

23

---

## Single cycle control:  inputs & outputs

**Step 1:**
Identify inputs & outputs

**Control Inputs:**
- **Opcode (6 bits)**
- **How about R-type instructions?**

} columns

**Control Outputs:**
- **RegDst**
- **ALUSrc**
- **MemtoReg**
- **RegWrite**
- **MemRead**
- **MemWrite**
- **Branch**
- **Jump**
- **ALUctr**

} rows

**Step 2:**
Make a control signal table for each cycle

24

# Control Signal Table

R-type (inputs)

(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
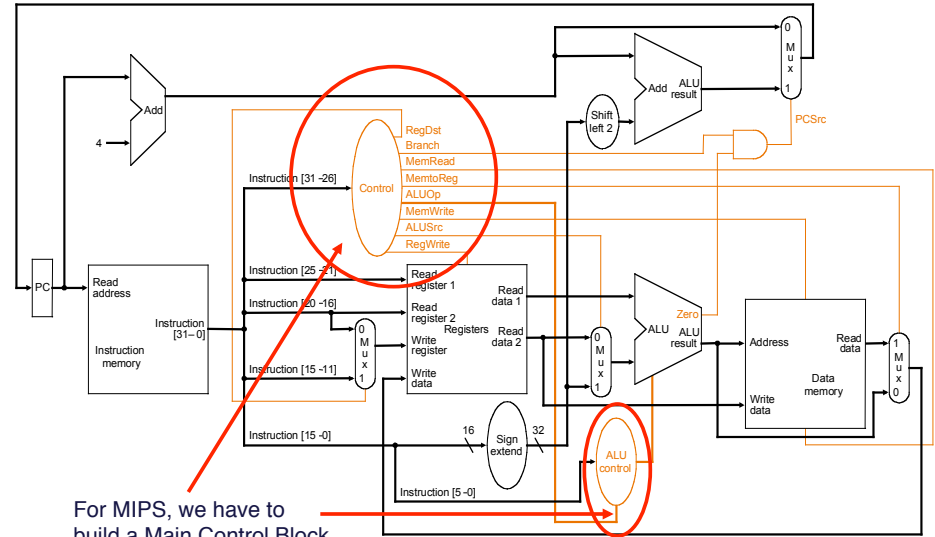


For MIPS, we have to build a Main Control Block and an ALU Control Block

# Main control, ALU control



Our 2 blocks of control logic
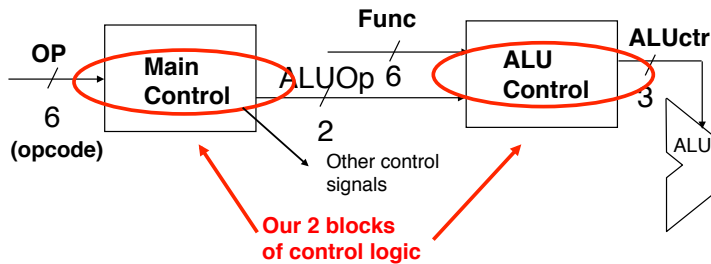
- **Use OP field to generate ALUOp (encoding)**
  - **Control signal fed to ALU control block**
- **Use Func field and ALUOp to generate ALUctr (decoding)**
  - **Specifically sets 3 ALU control signals**
    - **B-Invert, Carry-in, operation**

# Main control, ALU control



Outputs of main control, become *inputs* to ALU control
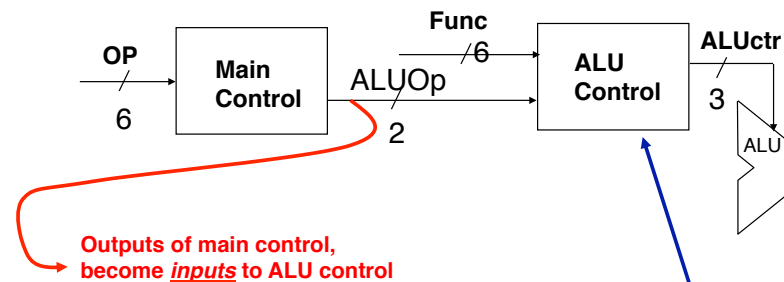
| | R-type | lw | sw | beq |
|---|---|---|---|---|
| ALU Operation | "R-type" | add | add | subtract |
| ALUOp<1:0> | 10 | 00 | 00 | 01 |

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 bits (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) |

29

# 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) |



Ex: ALUctr<2> (SUB/BEQ)
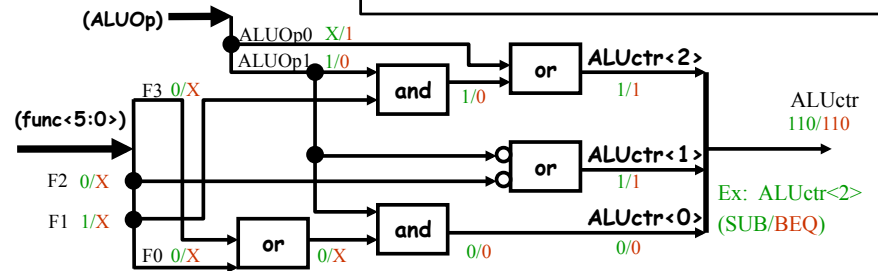
30

# The HW needed, plus control
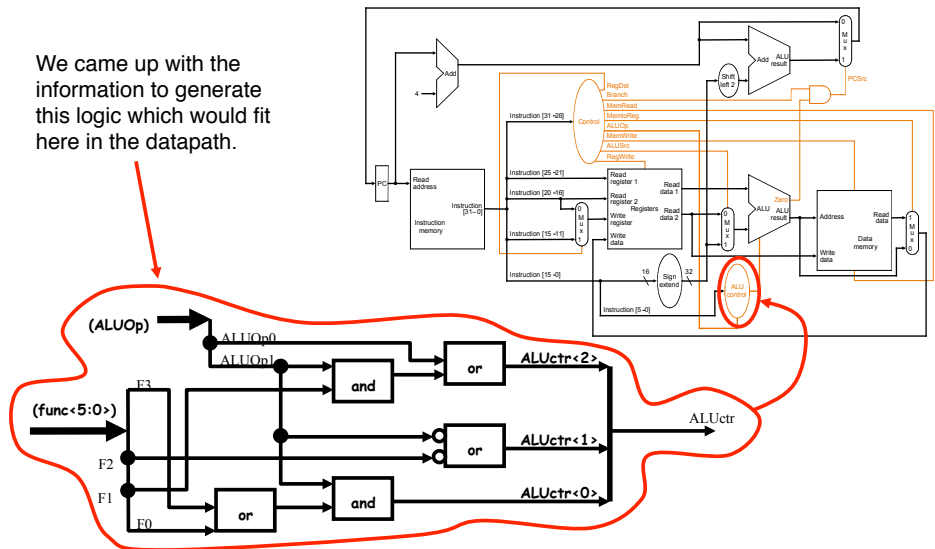
Single cycle MIPS machine
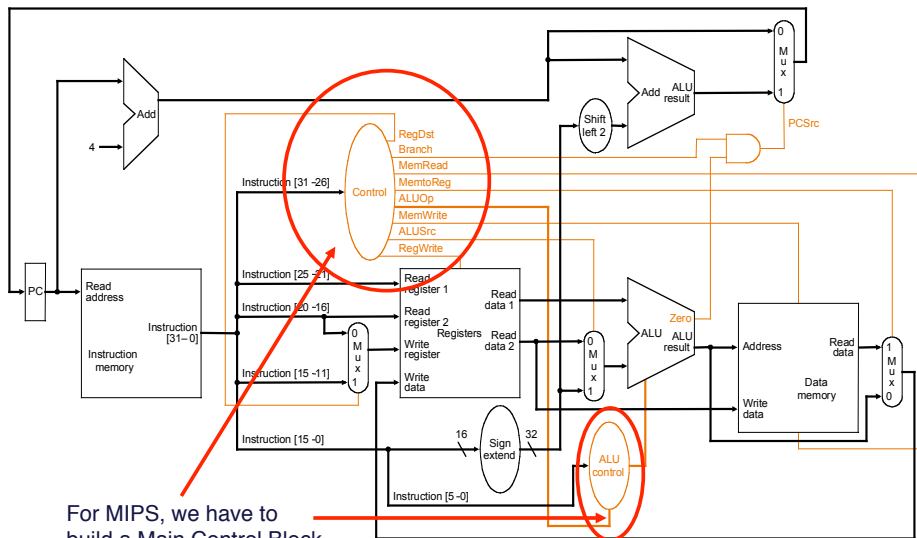


For MIPS, we have to build a Main Control Block and an ALU Control Block

31

# 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.
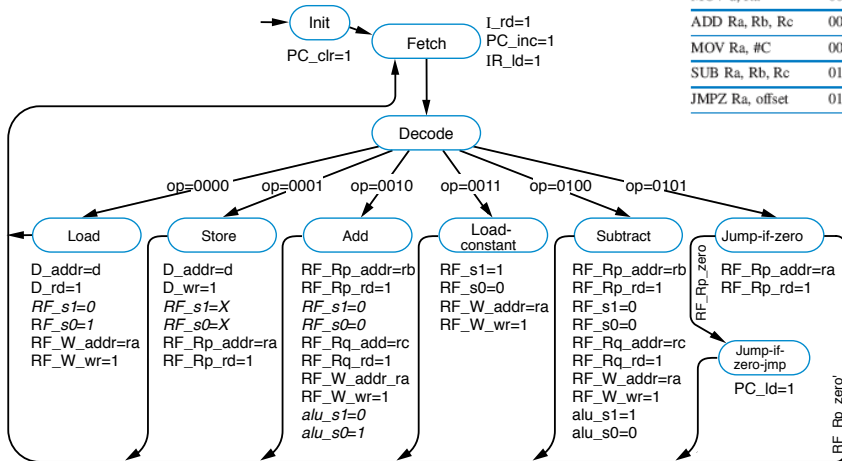


32

# Controller FSM for 6-instruction processor

**Recall:**
- With multi-cycle, need to generate control signals at right time
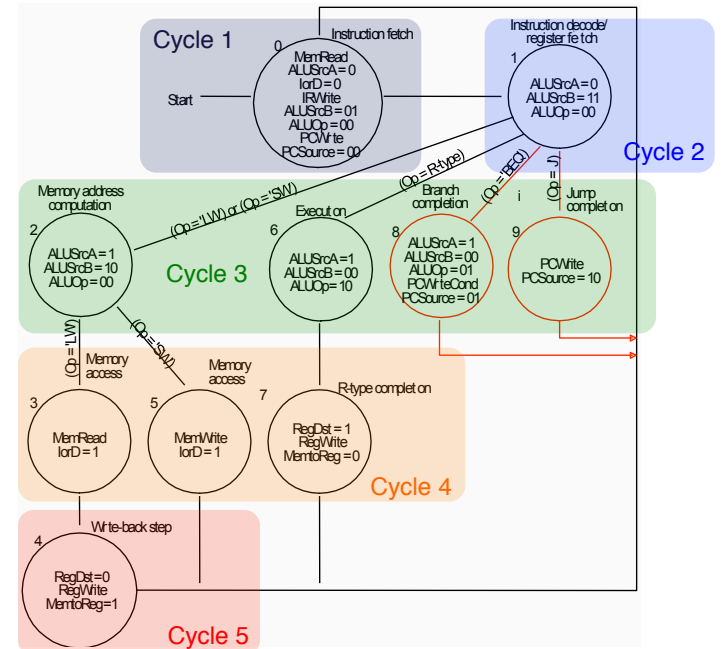- Captured by FSM
- Each level analogous to 1 CC

TABLE 8.2 Instruction opcodes.

| Instruction | Opcode |
|---|---|
| MOV Ra, d | 0000 |
| MOV d, Ra | 0001 |
| ADD Ra, Rb, Rc | 0010 |
| MOV Ra, #C | 0011 |
| SUB Ra, Rb, Rc | 0100 |
| JMPZ Ra, offset | 0101 |



**Init** → PC_clr=1

**Fetch** → I_rd=1, PC_inc=1, IR_ld=1

**Decode**

op=0000 → **Load**
D_addr=d
D_rd=1
RF_s1=0
RF_s0=1
RF_W_addr=ra
RF_W_wr=1

op=0001 → **Store**
D_addr=d
D_wr=1
RF_s1=X
RF_s0=X
RF_Rp_addr=ra
RF_Rp_rd=1

op=0010 → **Add**
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_add=rc
RF_Rq_rd=1
RF_W_addr_ra
RF_W_wr=1
alu_s1=0
alu_s0=1

op=0011 → **Load-constant**
RF_s1=1
RF_s0=0
RF_W_addr=ra
RF_W_wr=1

op=0100 → **Subtract**
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_addr=rc
RF_Rq_rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s1=1
alu_s0=0

op=0101 → **Jump-if-zero**
RF_Rp_addr=ra
RF_Rp_rd=1

**Jump-if-zero-jmp**
PC_ld=1

33

# MIPS FSM diagram



Cycle 1 — Instruction fetch
Start
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

Cycle 2 — Instruction decode/register fetch
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Cycle 3 — Memory address computation (Op='LW') or (Op='SW')
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

Execution (Op=R-type)
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

Branch completion (Op='BEQ')
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

Jump completion (Op='J')
PCWrite
PCSource = 10

Cycle 4 — Memory access (Op = LW)
MemRead
IorD = 1

Memory access (Op = SW)
MemWrite
IorD = 1

R-type completion
RegDst = 1
RegWrite
MemtoReg = 0

Cycle 5 — Write-back step
RegDst=0
RegWrite
MemtoReg=1

34

# Might also store control signals in ROM



**Control Logic (in ROM)**

**Output**
PCWrite
PCWriteCond
IorD
MemRead
MemWrite
IRWrite
MemtoReg
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
NS3
NS2
NS1
NS0

**Input**

Op5 Op4 Op3 Op2 Op1 Op0

S3 S2 S1 S0

State register

**Instruction Register Opcode Field**
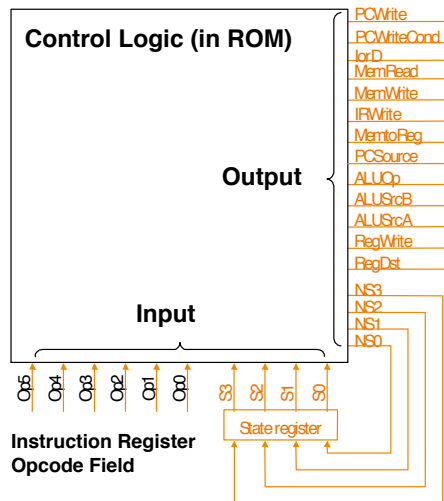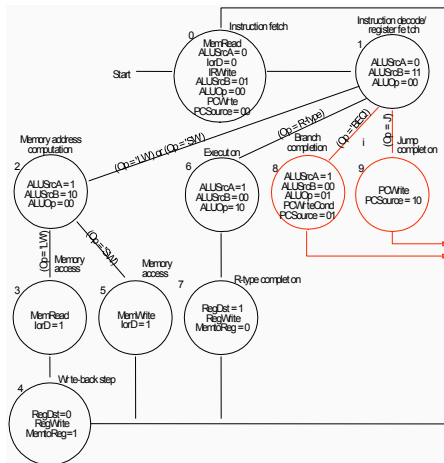
35

# 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**

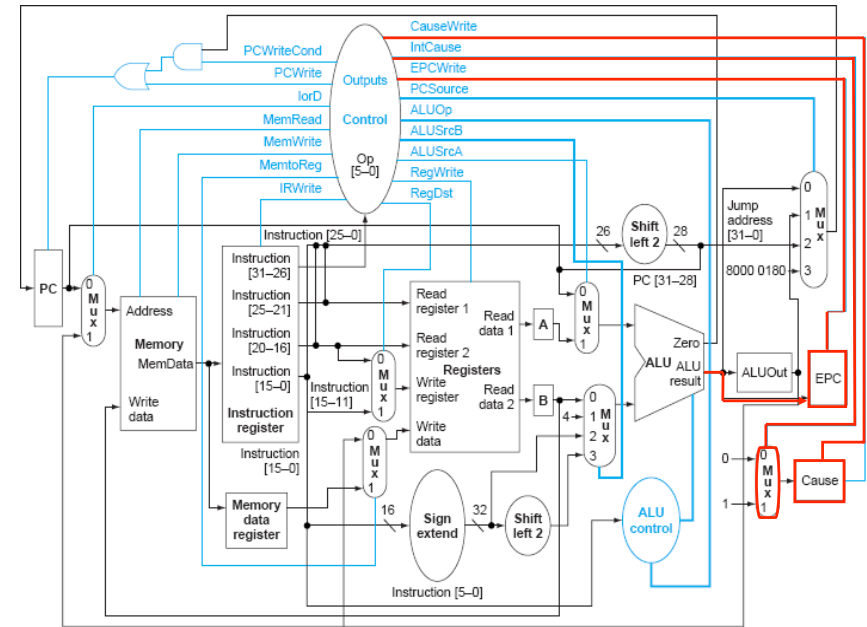**Another reason that register naming conventions are important.**

36

# FSM with Exception Handling
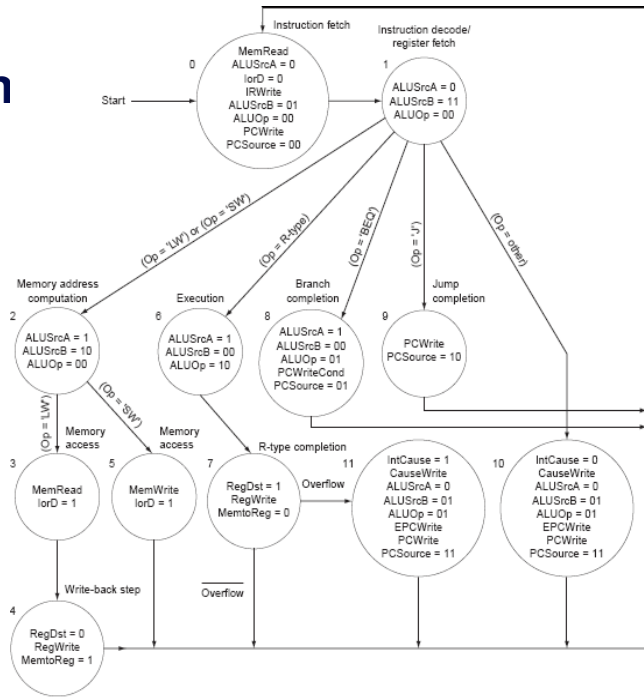
# Datapath with Exception Handling