

Review: Computer Organization

The Processor: Datapath and Control

Chansu Yu

Cleveland State University

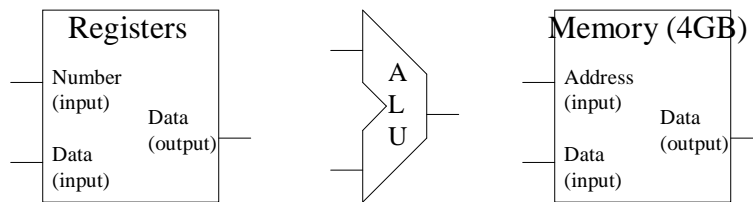
Implementation of the MIPS

- ❑ We're ready to look at an implementation of the MIPS
- ❑ Simplified to contain only:
 - memory-reference instructions: `lw`, `sw`
 - arithmetic-logical instructions: `add`, `sub`, `and`, `or`, `slt`
 - control flow instructions: `beq`, `j`
- ❑ Generic Implementation:
 - use the program counter (PC) to supply instruction address
 - get the instruction from memory
 - read registers
 - use the instruction to decide exactly what to do

Datapath of a Processor

□ Datapath

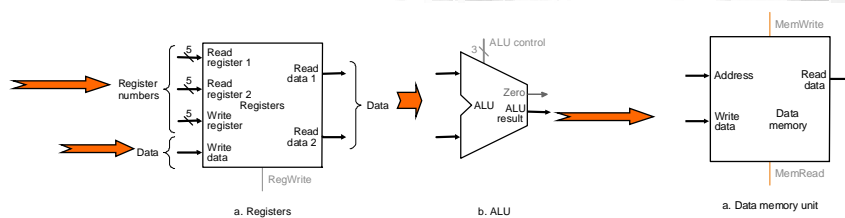
- All necessary data connection among the building blocks



- How many bits for each connection ?

3

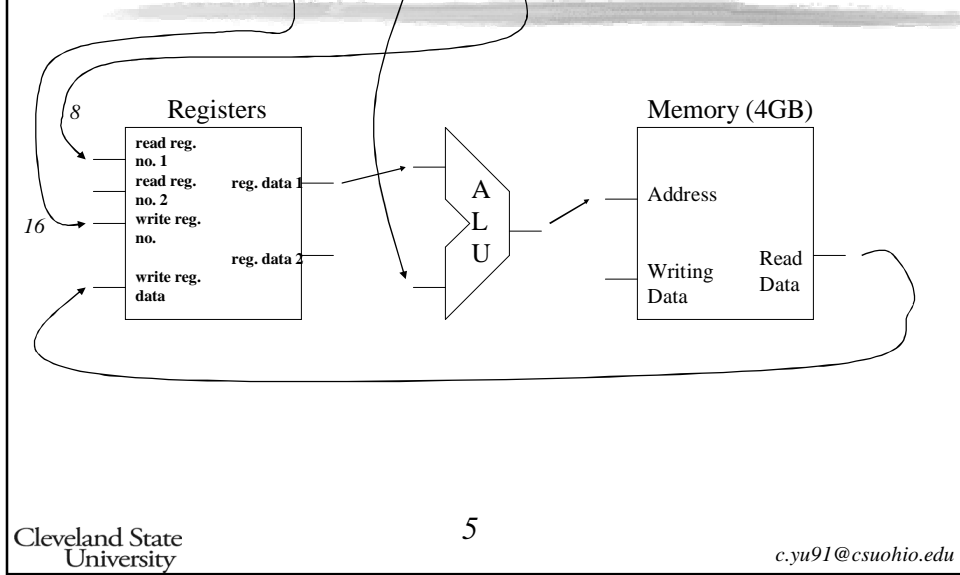
Register File, ALU and Memory



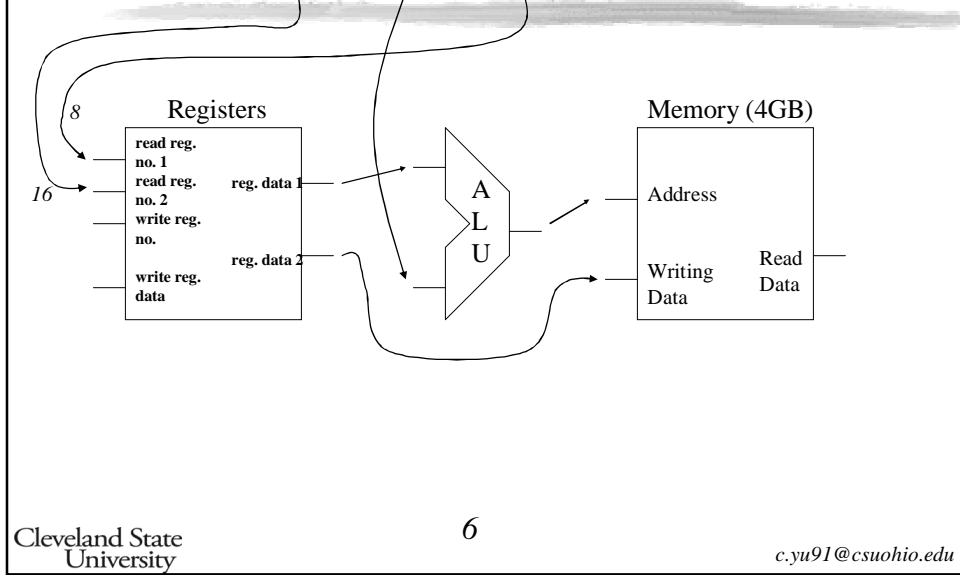
- Why register block has three inputs for register number and two outputs for register data ?
- ALU output can be fed into
 - Register block as a data input : Why ? And why not register number input ?
 - Memory block as an address : Why ? And why not data input ?
- What if it is not load/store architecture
 - add \$s0, \$s1, 96(\$t0)

4

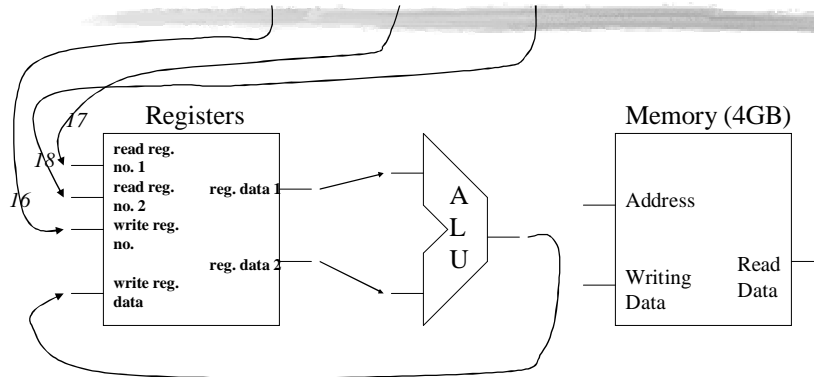
Ex: lw \$16, 100(\$8)



Ex: sw \$16, 100(\$8)

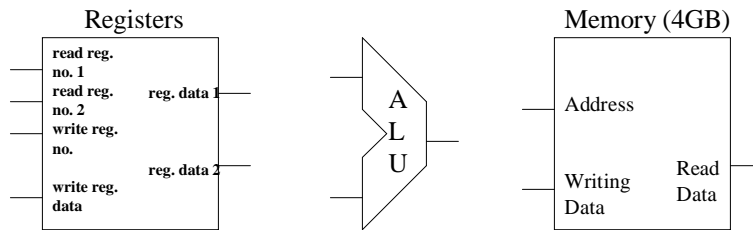


Ex: add \$16, \$17, \$18



Memory is "not" used at all !!!
 => Load/store architecture: memory access is allowed only on load/store instruction

Ex: add \$16, \$17, 100(\$18)



If the above instruction is a legitimate one, how to connect the blocks to implement it?

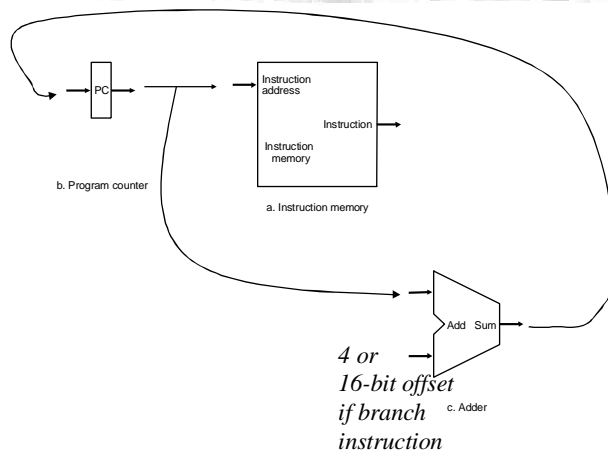
Input/Output for the Blocks

- ❑ Consider instructions: “previous four slides”
 - lw \$s0, 96(\$t0)
 - sw \$t3, 4(\$s2)
 - add \$s0, \$s1, \$s2

- ❑ MIPS addressing: “next slide”
 - $96(\$t0) = 96 + \$t0$
 - $\text{new } \$pc = \text{old } \$pc + 4$
 - $\text{new } \$pc = \text{old } \$pc + \text{immediate}$
(PC-relative addressing)

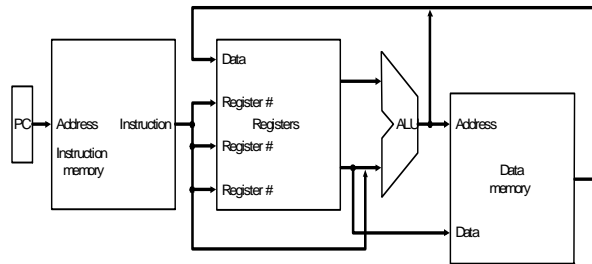
- ❑ All instructions use the ALU after reading the registers
Why? memory-reference? arithmetic? control flow?

Instruction Memory, PC, Adder



Implementation Details

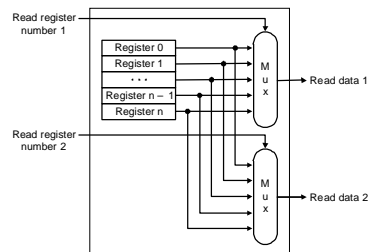
- Abstract / Simplified View:



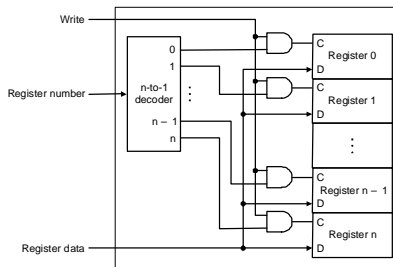
- Two types of functional units:
 - elements that operate on data values (combinational)
 - elements that contain state (sequential)

Register File

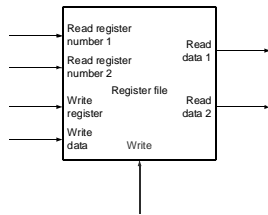
- Built using D flip-flops
(read)



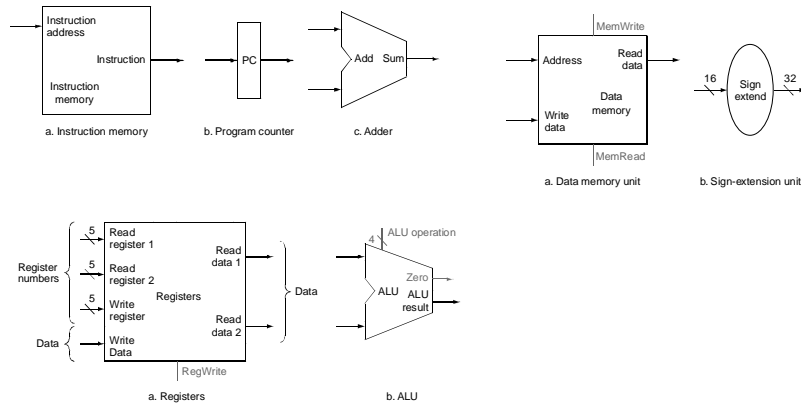
- (write)



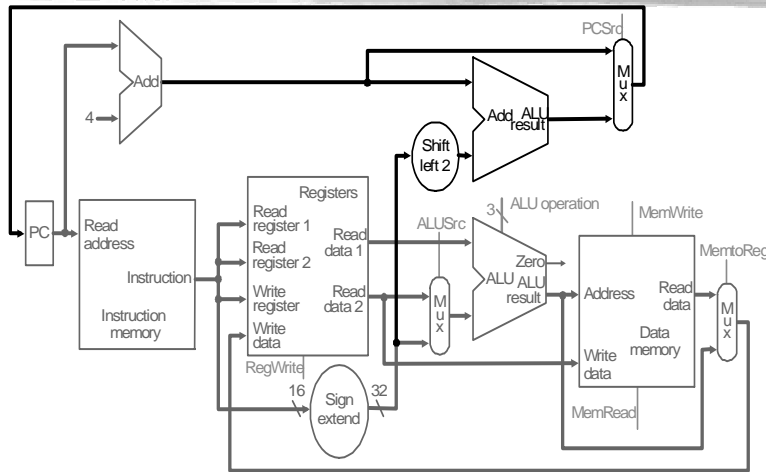
Register File



All Functional Units

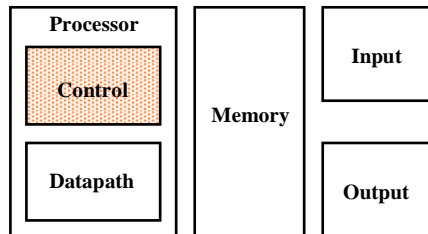


Datapath with Multiplexers



The Big Picture

❑ The Five Classic Components of a Computer



❑ Datapath & Control

Datapath and Control

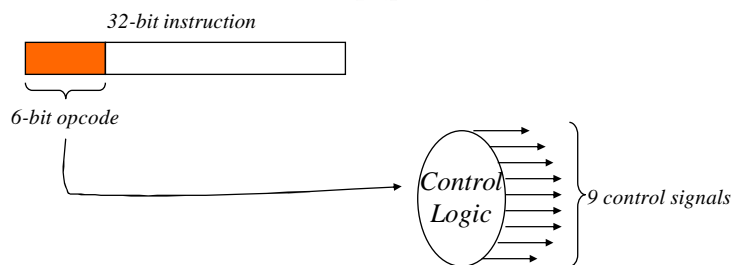
□ Datapath

- Between memory, ALU, register file
- When a 32-bit instruction is ready
 - It is decoded to obtain opcode, register numbers, ...
 - Register numbers go to register file and the register values become ready
 - Immediate value becomes ready
 - Memory data is read and ready

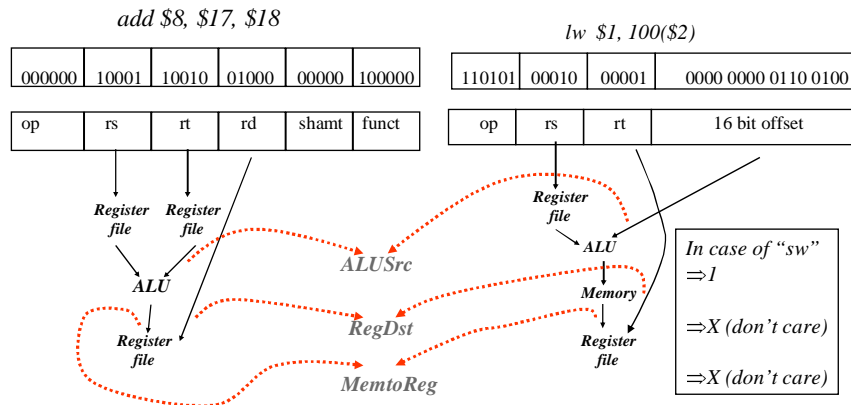
□ Control

- Need to decide which inputs should be used (multiplexors)
- ALU control such as operation, binvert, carryIn, ...
- Write signal (register write, memory write)

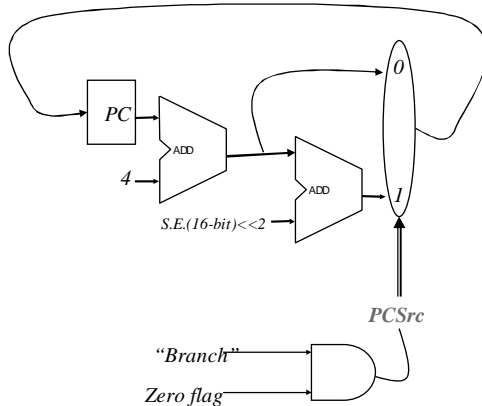
Generation of Control Signals



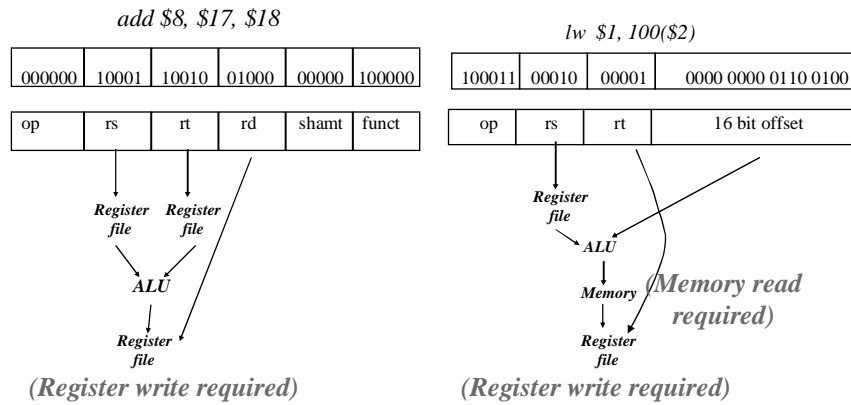
Multiplexer Selector



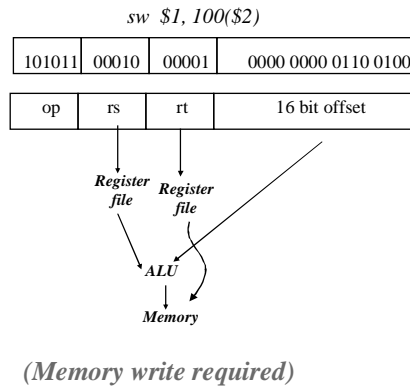
One More Multiplexor Selector



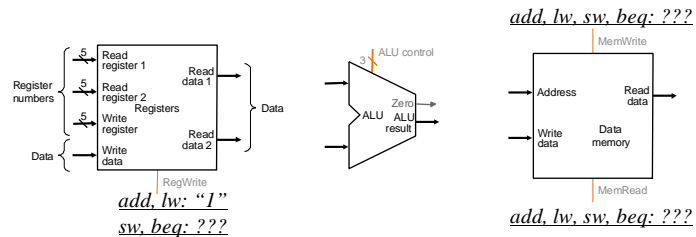
Write Signal



Write Signal



Write Signal



- What happen before write value to \$1 is ready ?**
- => **need some time to be stabilized**
 - => **this will be the longest path (critical path)**
 - => **determines the clock cycle for the CPU**

ALU Control

- ❑ What should the ALU do with this instruction ?
 - Information comes from the 32 bits of the instruction
 - ALU's operation based on instruction type and function code
- ❑ Multi-level control (for simplifying control logic)
 - Instruction's opcode (bit31-bit26)
 - => ALUOp1 & ALUOp0
(with instruction's funct (bit4-bit0))
 - => ALU inputs: binvert (= carryin), operation

Multicycle Approach

- ❑ 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

Breaking down an instruction

- ❑ ISA definition of arithmetic:

$$\text{Reg}[\text{Memory}[\text{PC}][15:11]] \leq \text{Reg}[\text{Memory}[\text{PC}][25:21]] \text{ op } \text{Reg}[\text{Memory}[\text{PC}][20:16]]$$

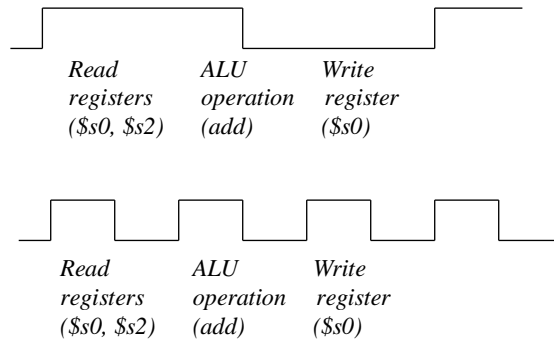
- ❑ Could break down to:

- $\text{IR} \leq \text{Memory}[\text{PC}]$
- $\text{A} \leq \text{Reg}[\text{IR}[25:21]]$
- $\text{B} \leq \text{Reg}[\text{IR}[20:16]]$
- $\text{ALUOut} \leq \text{A op B}$
- $\text{Reg}[\text{IR}[20:16]] \leq \text{ALUOut}$

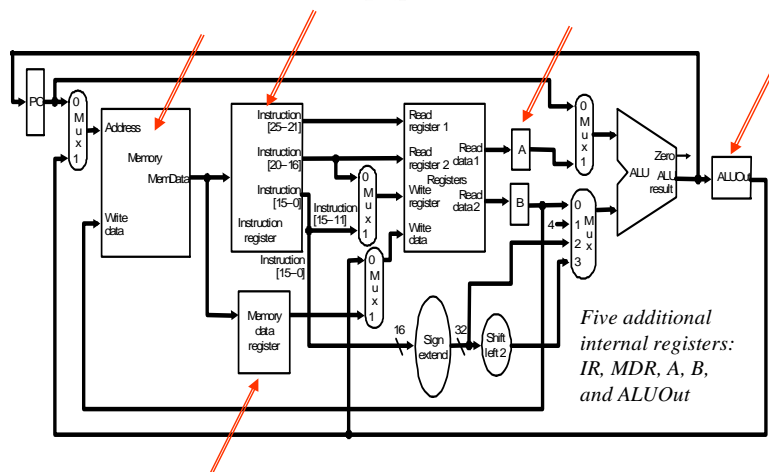
- ❑ We forgot an important part of the definition of arithmetic!

- $\text{PC} \leq \text{PC} + 4$

Comparison



Multicycle Approach



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Memory-read completion step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];  
PC = PC + 4;
```

For which kinds
of instructions?

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

➤ *"ALU will be busy calculating something else in other cycles"*

Step 2: Instruction Decode and Register Fetch

- ❑ Read registers *rs* and *rt* in case we need them
- ❑ Compute the branch address in case the instruction is a branch
- ❑ RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

For which kinds of instructions?

- ❑ We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ❑ ALU is performing one of three functions, based on instruction type
- ❑ Memory Reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

- ❑ R-type:

```
ALUOut = A op B;
```

For which kinds of instructions?

- ❑ Branch:

```
if (A==B) PC = ALUOut;
```

Step 4: R-type Completion or Memory Access

- ❑ Loads and stores access memory

MDR = Memory[ALUOut];
or
Memory[ALUOut] = B;

For which kinds of instructions?

- ❑ R-type instructions finish

Reg[IR[15-11]] = ALUOut;

The write actually takes place at the end of the cycle on the edge

Step 5: Memory Read Completion

- ❑ Reg[IR[20-16]] = MDR;

For which kinds of instructions?

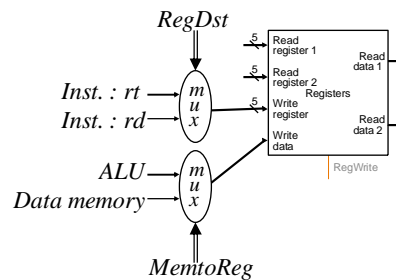
What about all the other instructions?

Which operations do we need ?

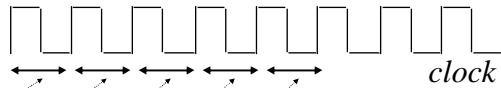
Set "MemtoReg" to "1"

Assert "RegWrite" signal

Set "RegDst" to "0"



Summary:



Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch		IR = Memory[PC] PC = PC + 4		
Instruction decode/register fetch		A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUOut = PC + (sign-extend(IR[15-0]) << 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	Reg[IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Implementing the Control

- ❑ Value of control signals is dependent upon:
 - what instruction is being executed
 - which step is being performed

- ❑ Use the information we've accumulated to specify a finite state machine
 - specify the finite state machine graphically, or
 - use microprogramming

- ❑ Implementation can be derived from specification
 - PLA (Appendix)
 - ROM (Appendix)
 - Microprogramming (Section 5.6)

5.6 Exceptions

- ❑ The hardest part of control is implementing exceptions and interrupts
 - Exception: unexpected event from within the processor (e.g., arithmetic overflow)
 - Interrupt: unexpected change of control flow caused by outside event (e.g., I/O communication with CPU)
- ❑ 2 types of exceptions covered here
 - Undefined instruction
 - Arithmetic overflow

Exception Type (all others)

0	INT	external interrupt
4	ADDRL	address error (load or instruction fetch)
5	ADDRS	address error (store)
6	IBUS	bus error on instruction fetch
7	DBUS	bus error on data load/store
8	SYSCALL	syscall exception
9	BKPT	breakpoint exception
<u>10</u>	<u>RI</u>	<u>reserved instruction exception</u>
<u>12</u>	<u>OVF</u>	<u>arithmetic overflow exception</u>

Including Exception

Additional input?
- Exception information

