CS/COE0447: Computer Organization and Assembly Language

Datapath and Control

Sangyeun Cho

Dept. of Computer Science
University of Pittsburgh

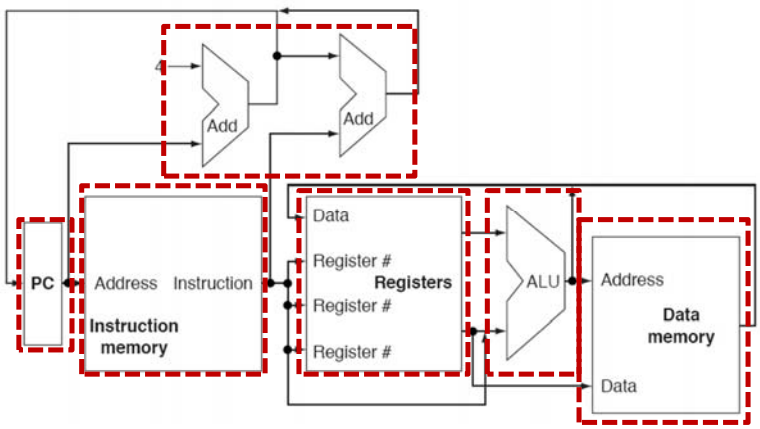
A simple MIPS

- We will design a simple MIPS processor that supports a small instruction set with…
- Memory access instructions
  - `lw` (load word) and `sw` (store word)
- Arithmetic-logic instructions
  - `add`, `sub`, `and`, `or`, and `slt`
- Control-transfer instructions
  - `beq` (branch if equal)
  - `j` (unconditional jump)

Initial ideas

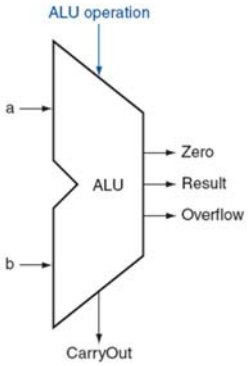
- Send program counter (PC) to code memory and fetch an instruction from there
  - PC keeps the address (i.e., pointer) where the target instruction is
- Read one or two registers
  - Depending on the instruction we fetched, we need to perform different actions with the values read from the register file
- Instructions of the same type (e.g., arithmetic-logic) will incur a similar sequence of actions

Abstract implementation



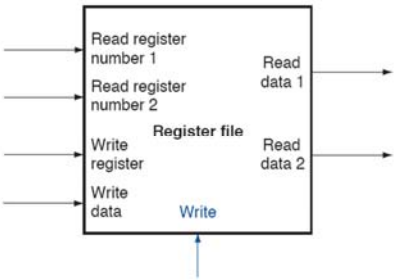
Datapath elements

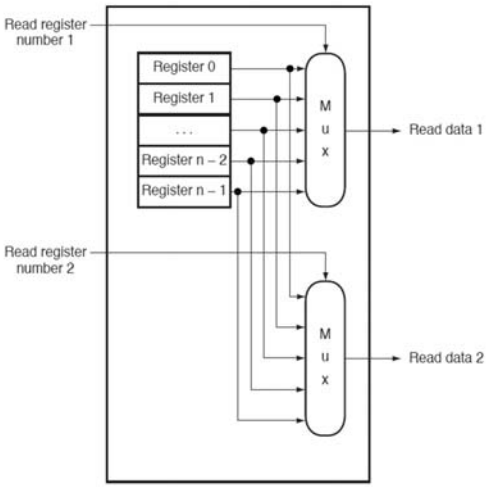
- Arithmetic logic unit (ALU)
  - Combinational logic (=function)
    - Input: a, b, ALU operation (carryin is hidden)
    - Output: result, zero, overflow, carryout
- Adders
  - For PC incrementing, branch target calculation, …
- Mux
  - We need a lot of these
- Registers
  - Register file, PC, … (architecturally visible registers)
  - Temporary registers to keep intermediate values



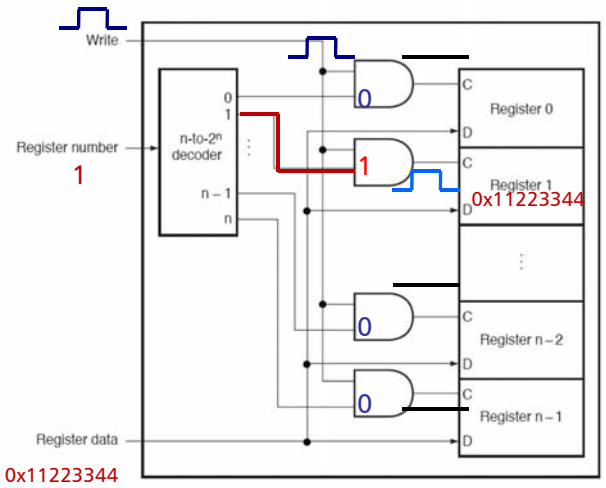
Register file

- Interface: read port, write port, clock, control signal

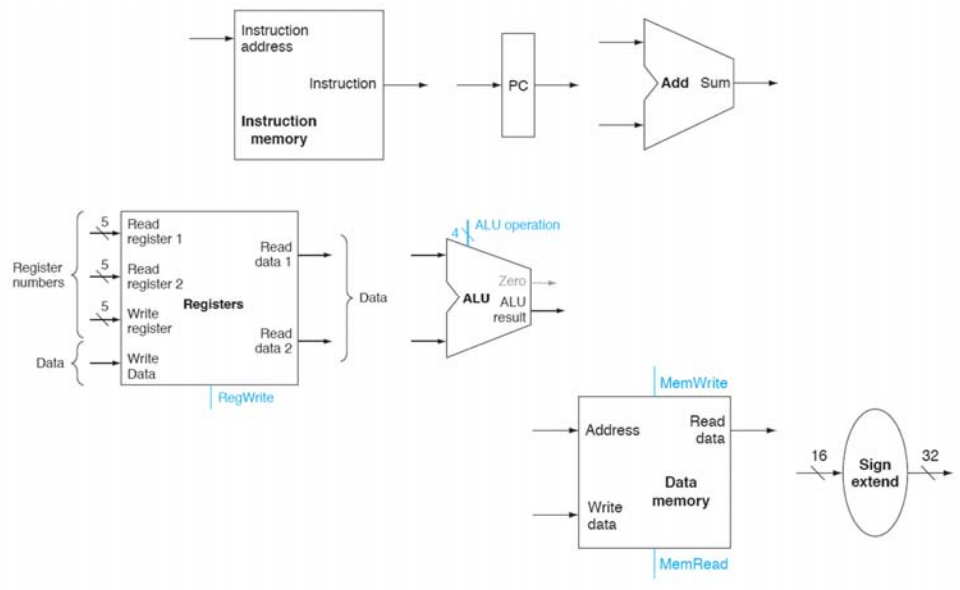




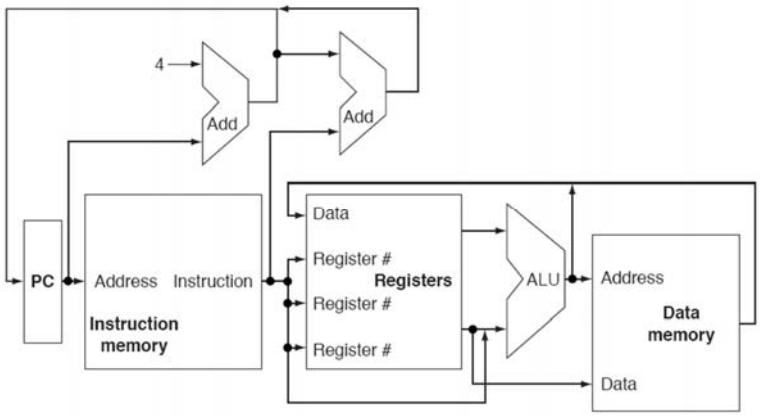
Register file



Processor building blocks



Abstract implementation



Analyzing instruction execution

- lw (load word)
  - Fetch instruction
  - Read a base register
  - Sign-extend the immediate offset
  - Add the two numbers made available in the above two steps
  - Access data memory with the address computed in the above step
  - Store the value from the memory to the target register specified in the instruction

Analyzing instruction execution

- add (add)
  - Fetch instruction
  - Read from two source registers
  - Add the two numbers made available in the above step
  - Store the result to the target register specified in the instruction

Analyzing instruction execution

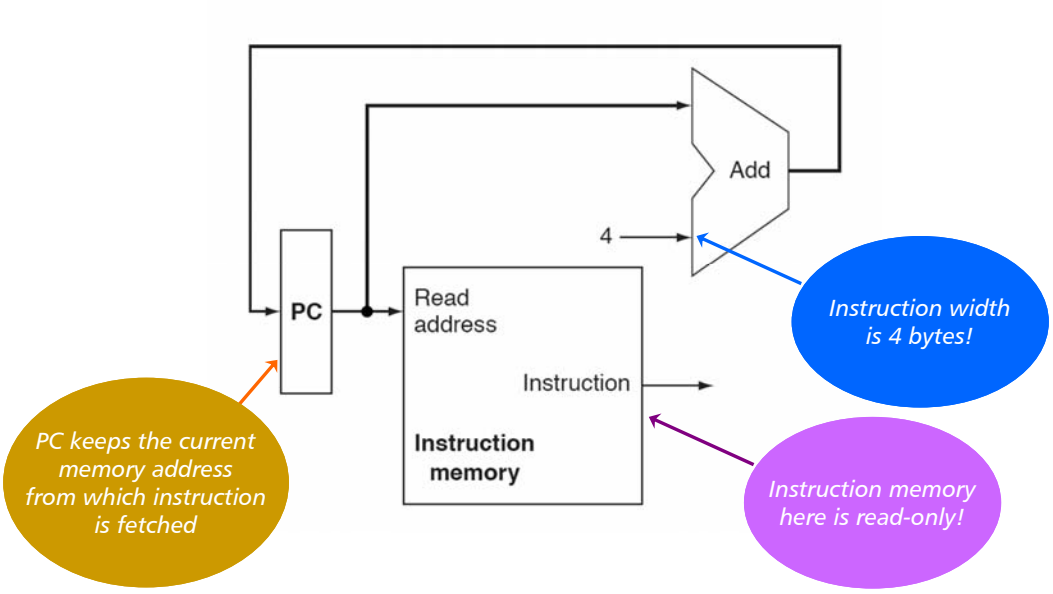
- j (jump)
  - Fetch instruction
  - Extend the 26-bit immediate field
    - Shift left by 2 bits (28 bits now)
    - Extract the most significant 4 bits from the current PC and concatenate to form a 32-bit value
  - Assign this value to PC

Common steps in inst. execution

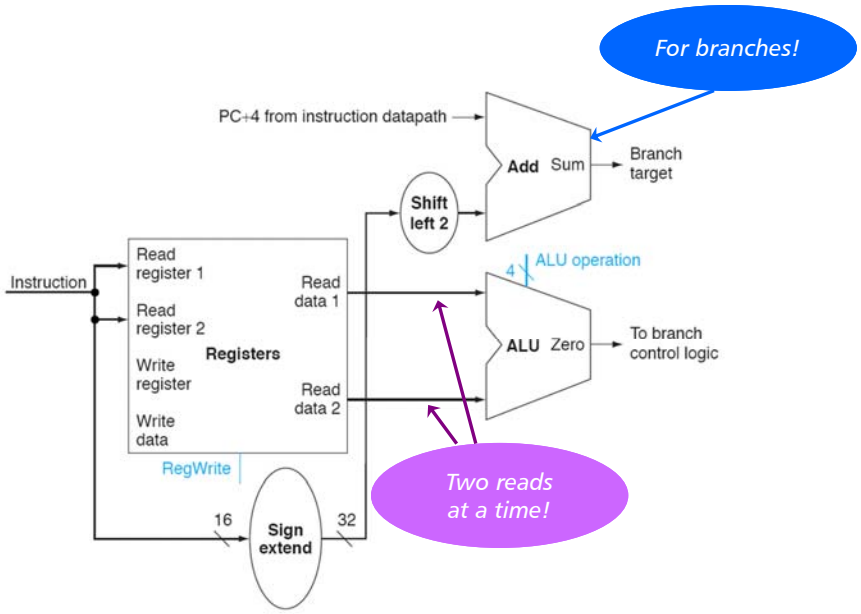
- Fetching the instruction word from the instruction memory
- Decoding the instruction and reading from the register file
  - Or prepare a value from the immediate value (and PC)
- Performing an ALU operation
- Accessing the data memory (if needed)
- Making a jump (assigning a computed value to PC) (if needed)
- Writing to the register file

- Designing a control logic is based on our (more formal) analysis of instruction execution
  - Consider all instructions

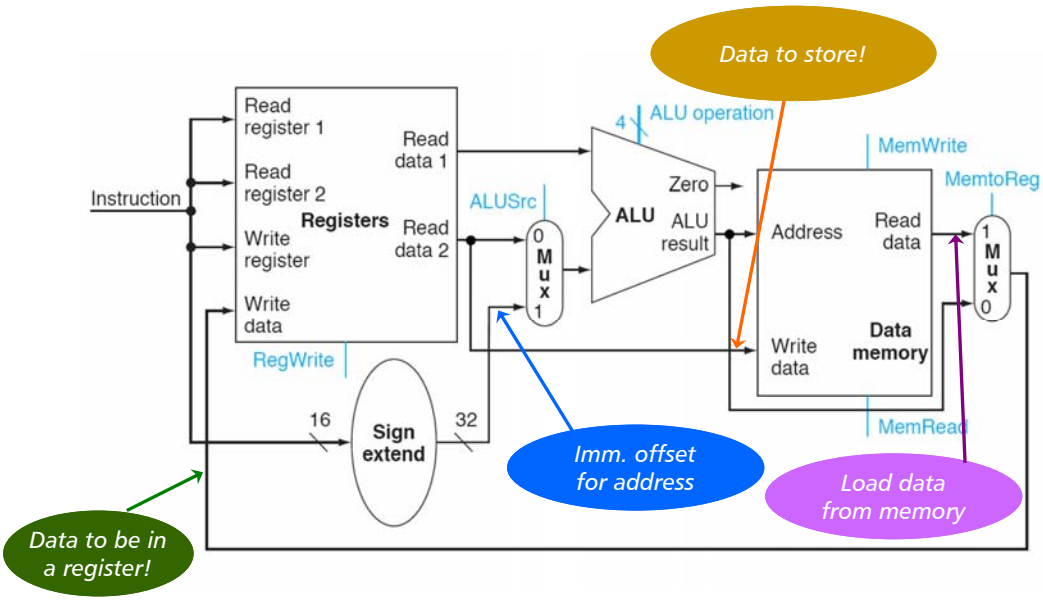
Fetching an instruction



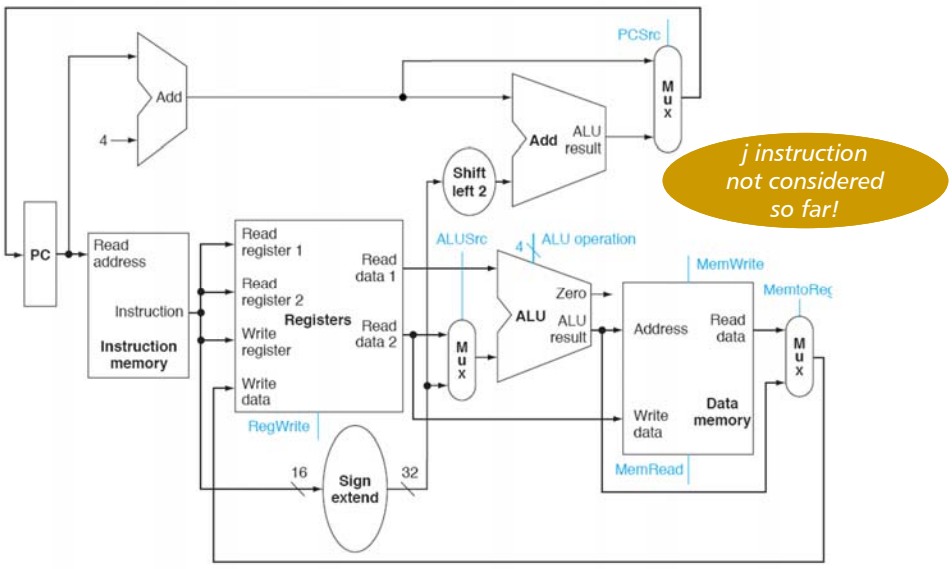
Fetching operands



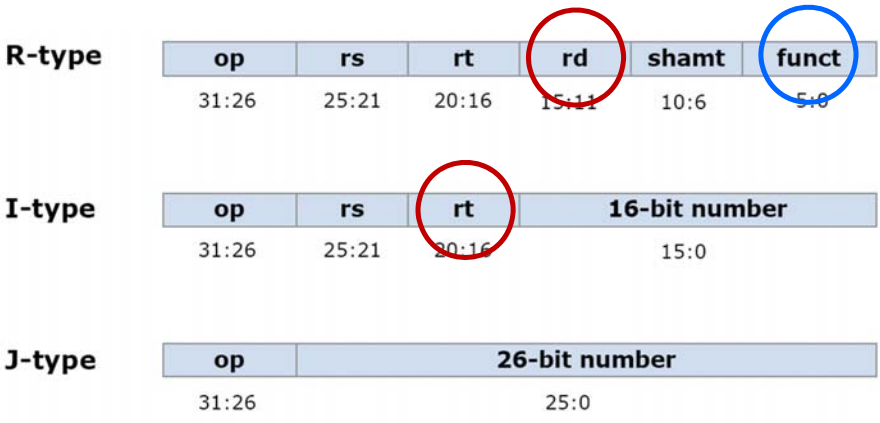
Handling memory access



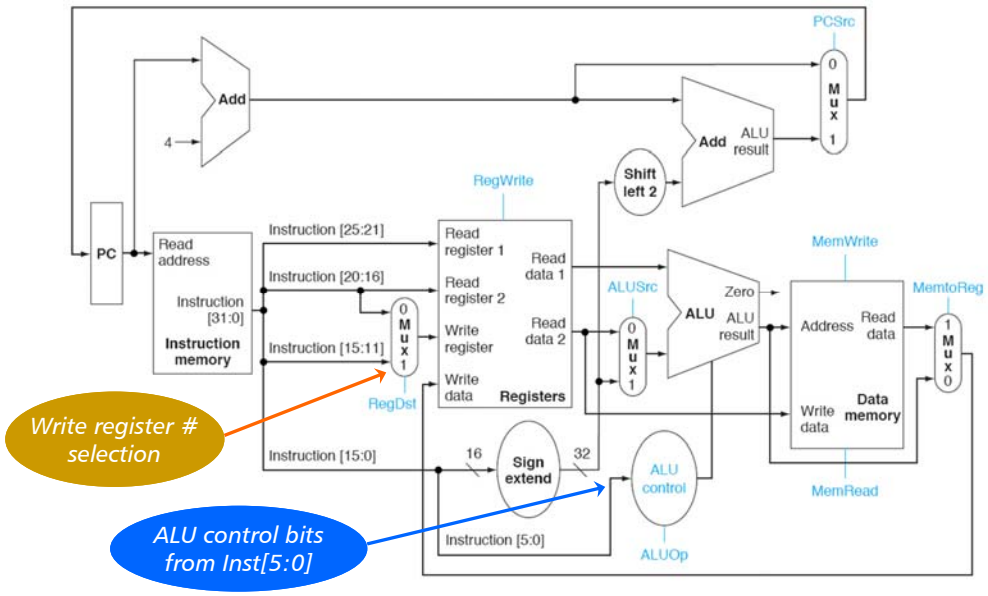
Datapath so far



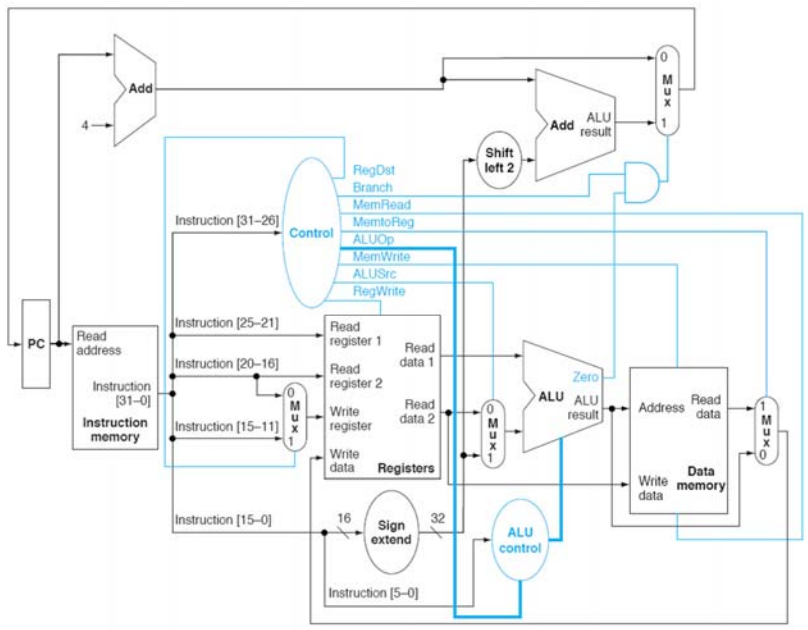
Revisiting MIPS inst. format



More elaborate datapath



First look at control



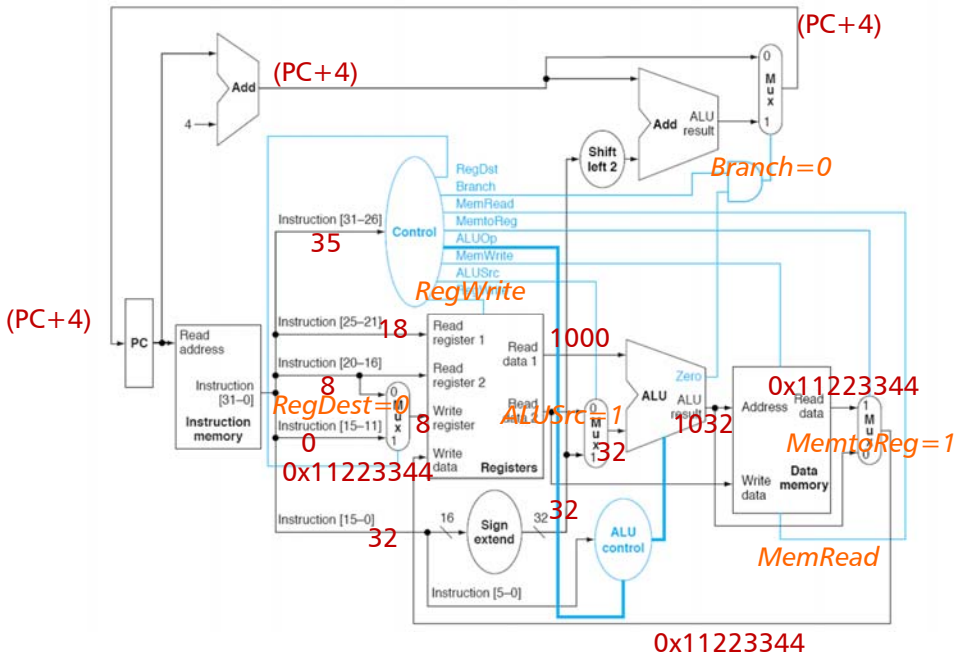
Control signals overview

- RegDst: which instr. field to use for dst. register specifier?
  - Inst[20:16] vs. Inst[15:11]
- ALUSrc: which one to use for ALU src 2?
  - Immediate vs. register read port 2
- MemtoReg: is it memory load?
- RegWrite: update register?
- MemRead: read memory?
- MemWrite: write to memory?
- Branch: is it a branch?
- ALUop: what type of ALU operation?

Example: lw r8, 32(r18)

- Let's assume r18 has 1,000
- Let's assume M[1032] has 0x11223344

Example: lw r8, 32(r18)



Control sequence for lw

- OP code=35

- RegDst=0
- ALUSrc=1
- MemtoReg=1
- RegWrite=1
- MemRead=1
- MemWrite=0
- Branch=0
- ALUop=0 (i.e., add) – why add?

Control signals in a table

Instruction	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0
LW	0	1	1	1	1	0	0	0	0
SW	X	1	X	0	0	1	0	0	0
BEQ	X	0	X	0	0	0	1	0	1

ALU control

- Depending on instruction, we perform different ALU operation
- Example
  - lw or sw: ADD
  - and: AND
  - beq: SUB
- ALU control input (3 bits)
  - 000: AND
  - 001: OR
  - 010: ADD
  - 110: SUB
  - 111: SET-IF-LESS-THAN (similar to SUB)

ALU control

- ALUop
  - 00: lw/sw, 01: beq, 10: arithmetic, 11: jump

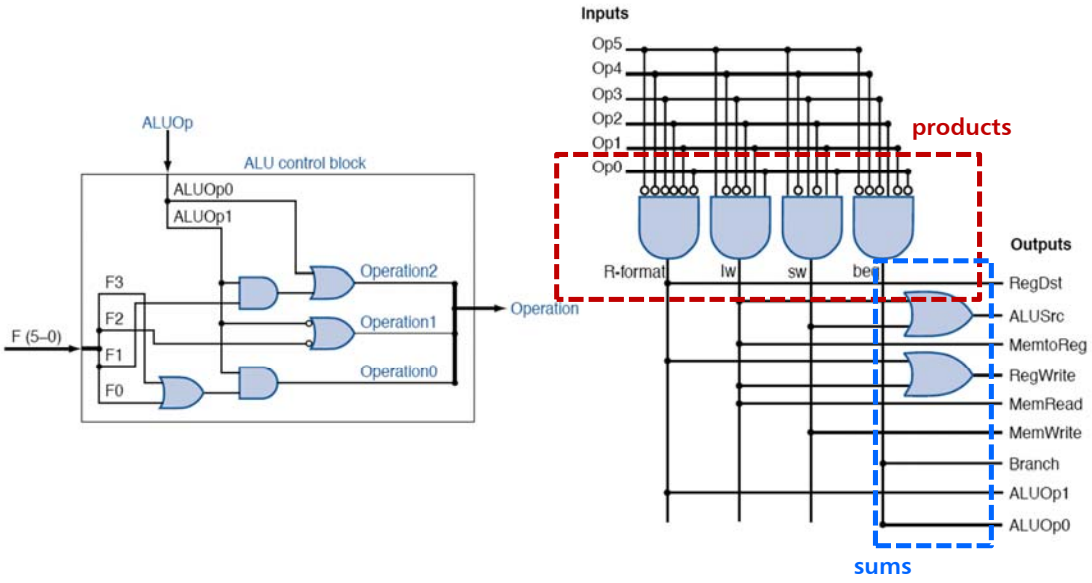
Instruction	ALUOp	Instruction operation	Funct Field	Desired ALU function	ALU control
LW	00	Load word	XXXXXX	Add	010
SW	00	Store word	XXXXXX	Add	010
BEQ	01	Branch if equal	XXXXXX	Subtract	110
R-type	10	ADD	100000	Add	010
R-type	10	SUB	100010	Subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	Set if less than	101010	Set if less than	111

ALU control truth table

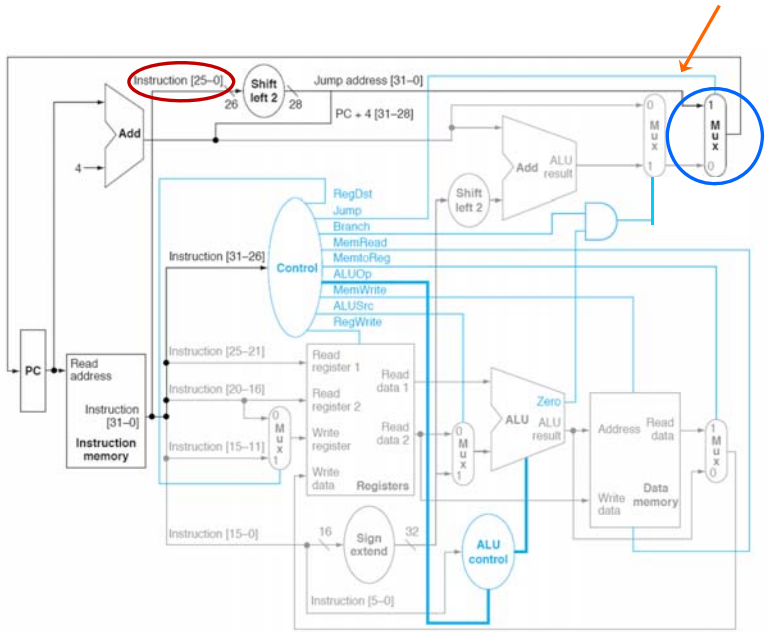
input output

ALUOp		Funct Field						ALU control
ALUOp1	**ALUOp0**	**F5**	**F4**	**F3**	**F2**	**F1**	**F0**	
0	0	X	X	X	X	X	X	010
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	0	1	0	0	0	0	0	010
1	0	1	0	0	0	1	0	110
1	0	1	0	0	1	0	0	000
1	0	1	0	0	1	0	1	001
1	0	1	0	1	0	1	0	111

ALU control logic implementation



Supporting "j" instruction



Resource usage

Instruction	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
LW	Instruction fetch	Register access	ALU	Memory access	Register access
SW	Instruction fetch	Register access	ALU	Memory access	
BEQ	Instruction fetch	Register access	ALU	0	0
Jump	Instruction fetch				

Single-cycle execution timing

Instruction class	Instruction memory	Register read	ALU operation	Data Memory	Register Write	Total
R-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
BEQ	200	50	100			350
Jump	200					200

(in pico-seconds)

Single-cycle execution problem

- The cycle time depends on the most time-consuming instruction
  - What happens if we implement a more complex instruction, e.g., a floating-point multiplication
  - All resources are simultaneously active – there is no sharing of resources
- We'll adopt a multi-cycle solution which allows us to
  - Use a faster clock;
  - Adopt a different number of clock cycles per instruction; and
  - Reduce physical resources

Multi-cycle implementation

- Reusing functional units
  - Break up instruction execution into smaller steps
  - Each functional unit is used for a specific purpose in any cycle
  - ALU is used for additional functions: calculation and PC increment
  - Memory used for instructions and data
- At the end of a cycle, keep results in registers
  - Additional registers
- Now, control signals are NOT solely determined by the instruction bits
- Controls will be generated by a FSM!

Five instruction execution steps

- Instruction fetch
- Instruction decode and register read
- Execution, memory address calculation, or branch completion
- Memory access or R-type instruction completion
- Write-back

- Instruction execution takes 3~5 cycles!

Step 1: instruction fetch

- Access memory w/ PC to fetch instruction and store it in Instruction Register (IR)

- Increment PC by 4 using ALU and put the result back in the PC
  - We can do this because ALU is not busy in this cycle
  - Actual PC Update is done at the next clock rising edge

Step 2: decode & operand fetch

- Read registers rs and rt
  - We read both of them regardless of necessity
- Compute the branch address using ALU in case the instruction is a branch
  - We can do this because ALU is not busy
  - *ALUOut* will keep the target address
- We have not set any control signals based on the instruction type yet
  - Instruction is being decoded now in the control logic!

Step 3: actions, actions, actions

- ALU performs one of three functions based on instruction type
- Memory reference
  - *ALUOut* <= *A* + sign-extend(*IR*[15:0]);
- R-type
  - *ALUOut* <= *A* op *B*;
- Branch:
  - if (*A*==*B*) *PC* <= *ALUOut*;
- Jump:
  - *PC* <= {*PC*[31:28],*IR*[25:0],2'b00}; // verilog notation

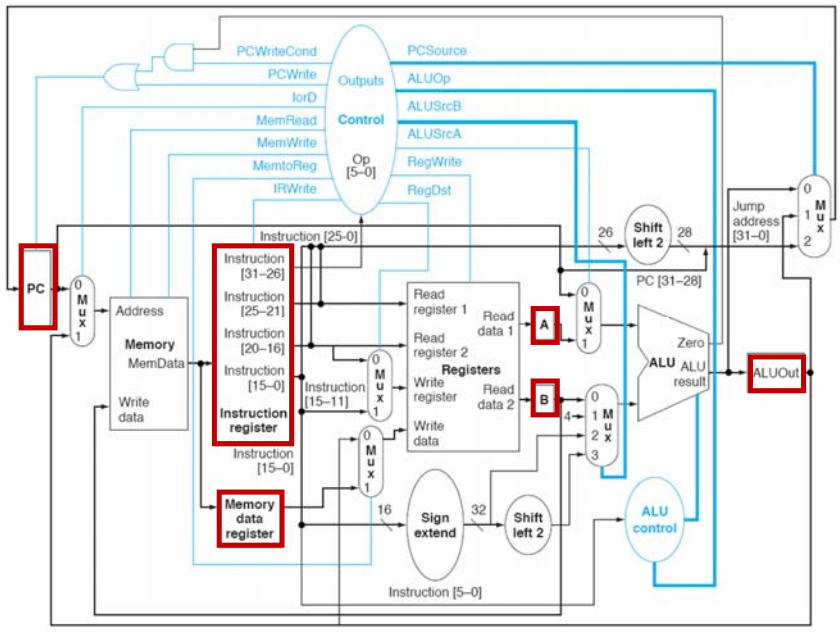
Step 4: memory access

- If the instruction is memory reference
  - *MDR* <= Memory[*ALUOut*]; // if it is a load
  - Memory[*ALUOut*] <= *B*; // if it is a store
  - Store is complete!
- If the instruction is R-type
  - Reg[*IR*[15:11]] <= *ALUOut*;
  - Now the instruction is complete!

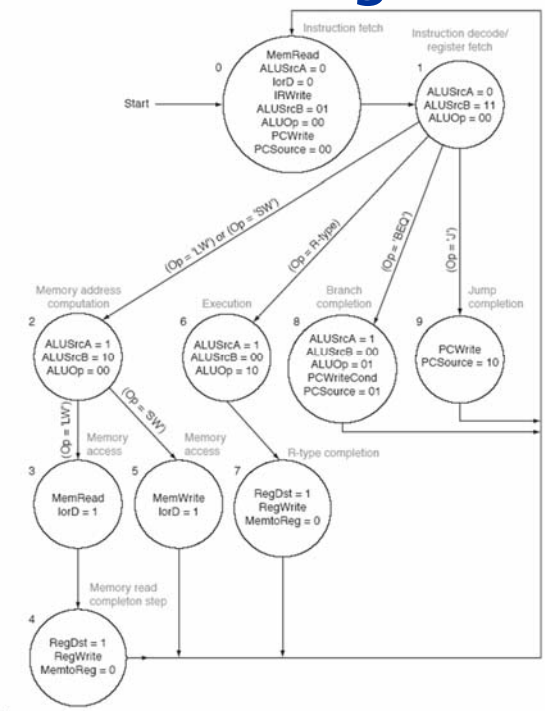
Step 5: register write-back

- Only memory load instruction reaches this step
  - Reg[*IR*[20:16]] <= *MDR*;

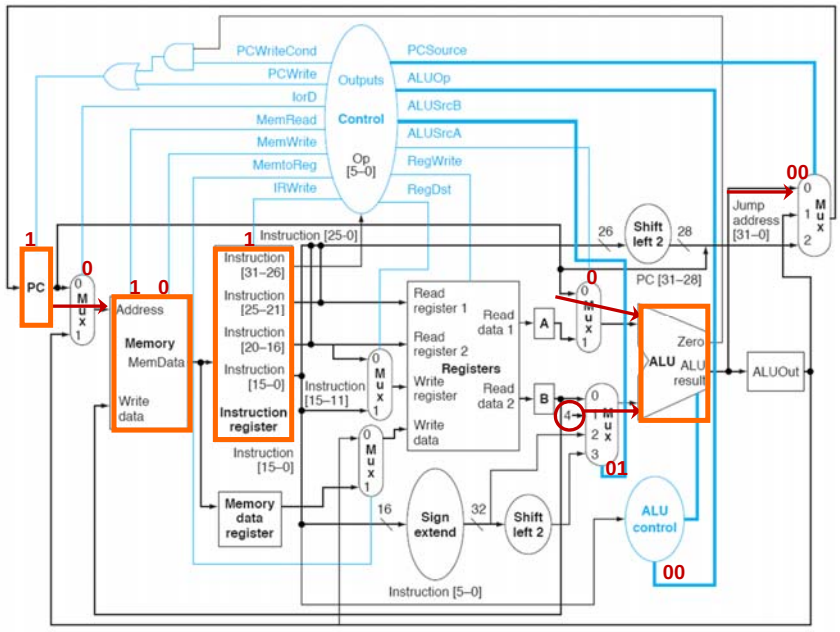
Multi-cycle datapath & control



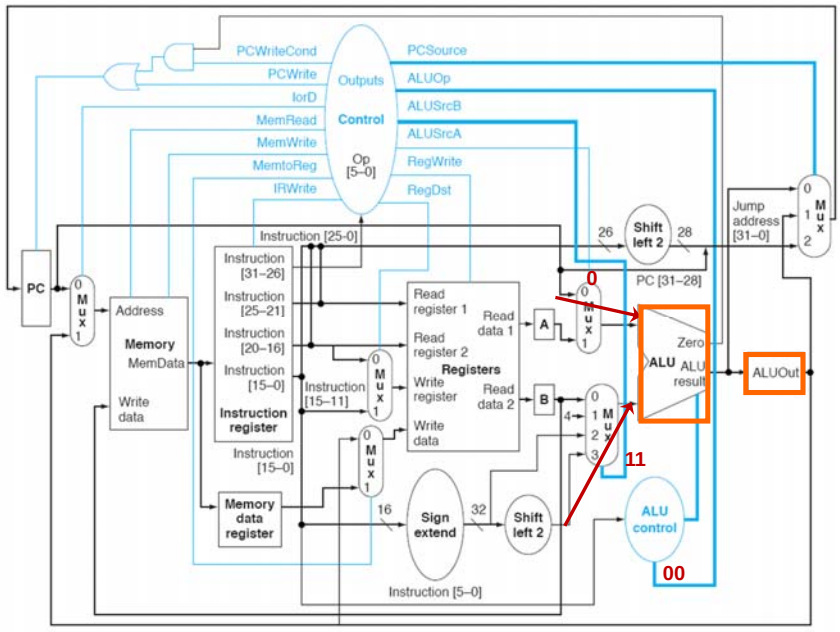
Multi-cycle control design



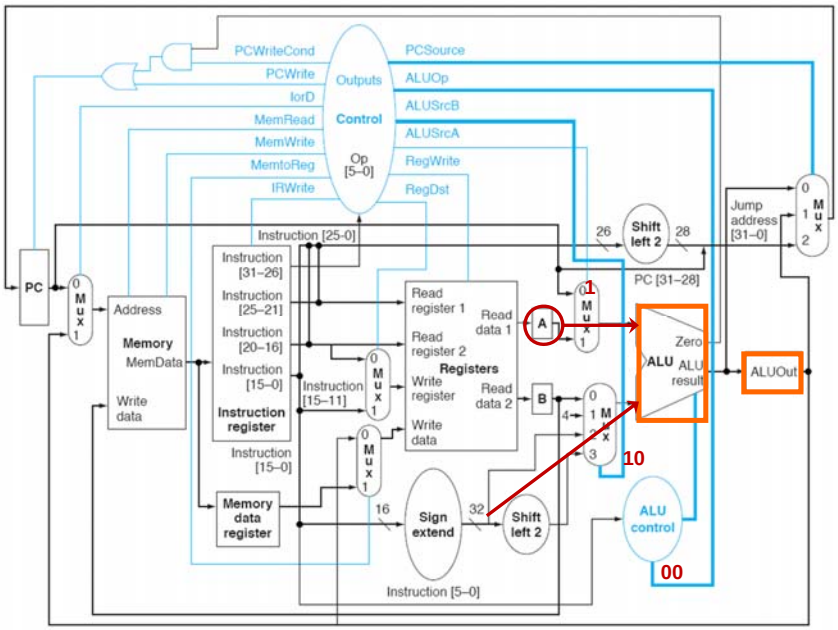
Example: lw, 1st cycle



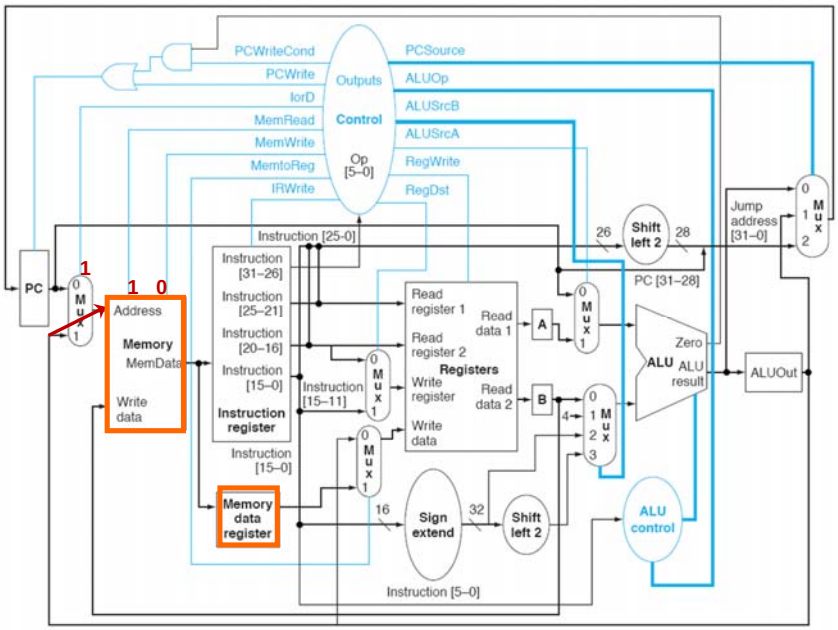
Example: lw, 2nd cycle



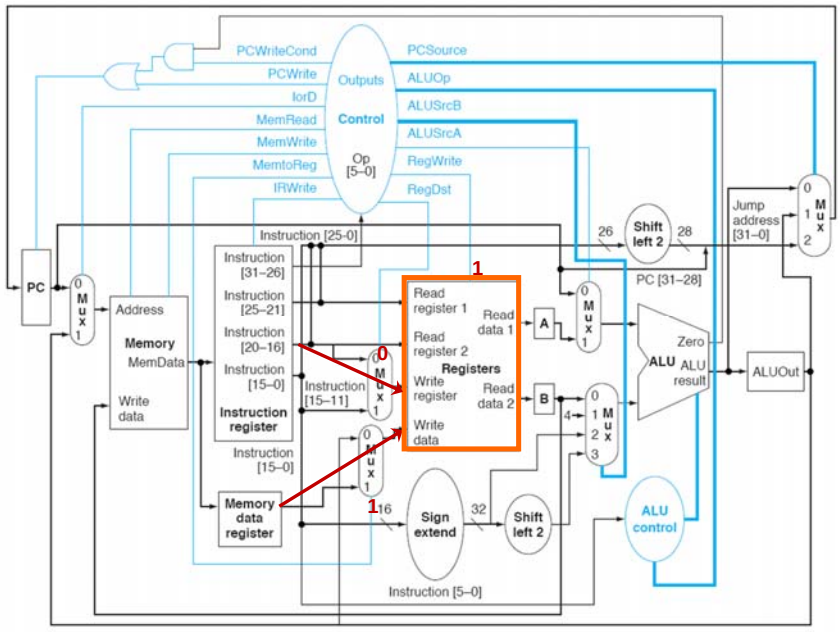
Example: lw, 3rd cycle



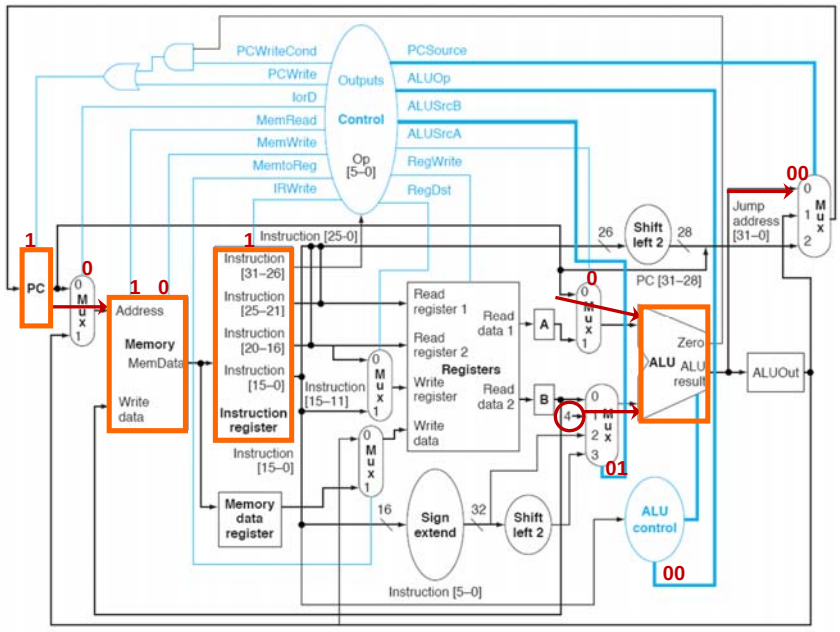
Example: lw, 4th cycle



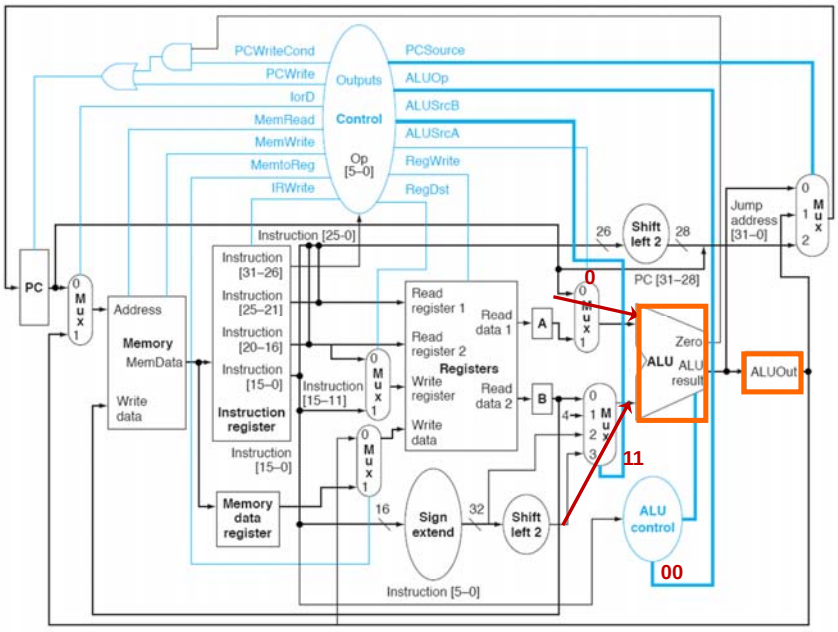
Example: lw, 5th cycle



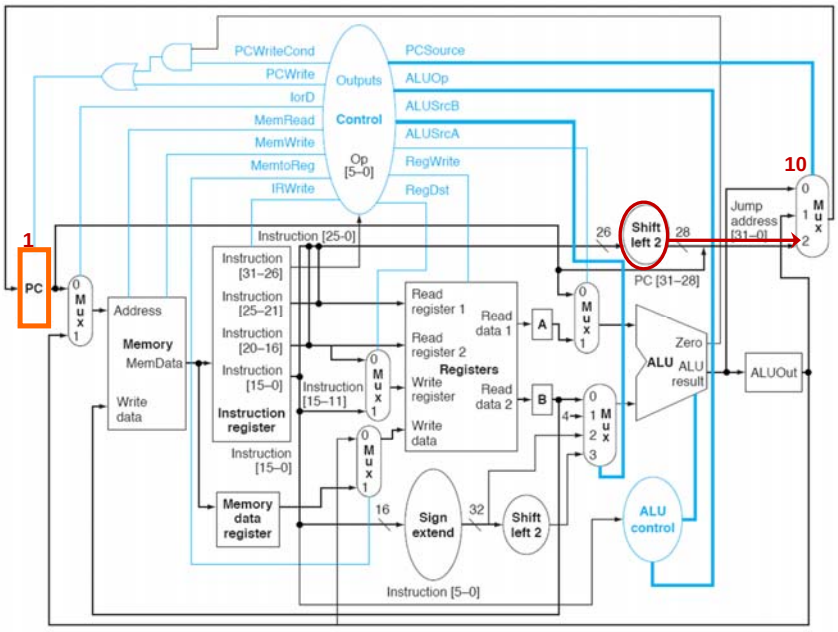
Example: j, 1st cycle



Example: j, 2nd cycle



Example: j, 3rd cycle



To wrap up

- From a number of building blocks, we constructed a datapath for a subset of the MIPS instruction set
- First, we analyzed instructions for functional requirements
- Second, we connected buildings blocks in a way that accommodates instructions
- Third, we kept refining the datapath

To wrap up

- We looked at how an instruction is executed on the datapath in a pictorial way
- Control signals were connected to functional blocks in the datapath
- How execution sequence of an instruction change the control signals was analyzed
- We looked at the multi-cycle control scheme in some detail
  - Multi-cycle control can be implemented using FSM

To wrap up

- We looked at the multi-cycle control scheme in some detail
- Multi-cycle control can be implemented using FSM