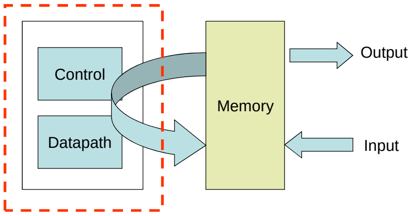
ECE437: Introduction to Digital Computer Design

Chapter 4a (single-cycle, 4.1-4.4)

Fall 2016

Processor Implementation



362 vs. 437

- 362 CPU is an embedded processor
  - Low cost, low power emphasis
  - On-chip, custom peripherals key
  - Transistors used for peripherals
- 437 CPU is a general-purpose processor
  - High performance emphasis
  - Off-chip, generic peripherals
  - Transistors used for performance
- You must have noticed differences in ISA

Outline

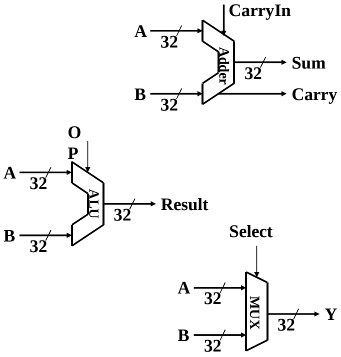
- Datapath - single cycle
  - single instruction, 2's complement, unsigned
- Control

Datapath for Instructions

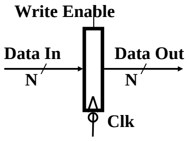
- Single-cycle datapath
- Compose using well-understood pieces
  - Mux, flip-flops and gates
  - ALU
  - Register file

Comb. Logic Elements

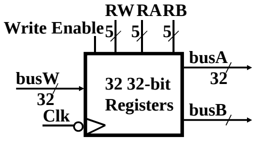
- Adder
- ALU
- Mux



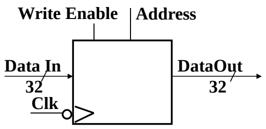
Storage Elements



- Register
  - for PC

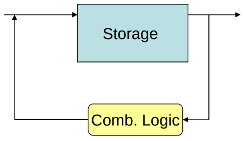


- Register file
  - 32 registers
  - 2 read ports/buses
  - 1 write port/bus



- Memory
  - 1 input bus
  - 1 output bus
  - Not bidirectional

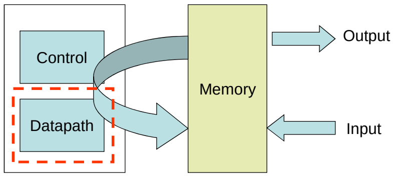
Computer as State Machine



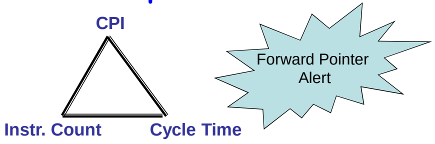
- Storage elements
  - Memory, Register file, PC
- Combinational elements
  - ALUs, Adders, Muxes

Processor Implementation

- This Lecture: Datapath



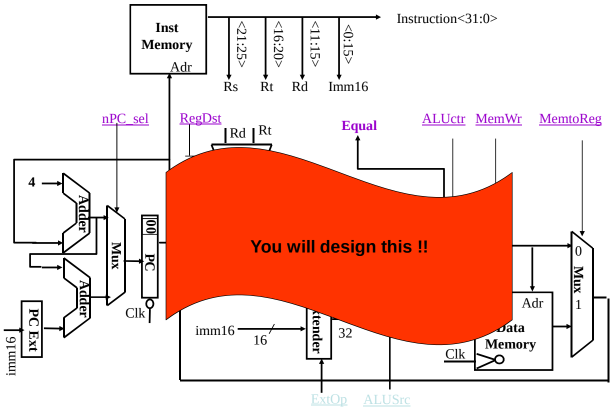
Processor Implementation



- Implementation determines
  - How many Clocks Per Instruction (CPI)
  - How long is the clock cycle (Cycle time)
- ISA, compiler determine
  - How many instructions in a program (Instr. count)
- We will cover these in Ch 1b (after Ch 4a)
  - For now: implementation determines execution time which measures performance

Datapath – Single cycle

- Assumption: Get one whole instruction done in one long clock cycle
  - fetch, decode/read operands, execute, memory, writeback
    - 5 steps you should NEVER forget!
  - useful way to represent steps and identify required datapath elements: RTL
- For single instruction
- Put it together



You will design this !!

Warning

- Processor you will design in the lab will NOT be exactly same as the processor described in lectures
- So blindly copying design on lecture slides WON'T work in the lab
- Helps me (and you) know whether you understand the material

Outline

- Datapath - single cycle
  - single instruction, 2's complement, unsigned
- Control

Register Transfer Language

- RTL gives the <u>meaning</u> of the instructions
- All start by fetching the instruction

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

inst	**Register Transfers**	
ADDU	**R[rd] <– R[rs] + R[rt];**	**PC <– PC + 4**
SUBU	**R[rd] <– R[rs] – R[rt];**	**PC <– PC + 4**
ORi	**R[rt] <– R[rs] OR zero_ext(Imm16);**	**PC <– PC + 4**
LOAD	**R[rt] <– MEM[R[rs] + sign_ext(Imm16)];**	**PC <– PC + 4**
STORE	**MEM[R[rs] + sign_ext(Imm16)] <– R[rt];**	**PC <– PC + 4**
BEQ	**if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4**	

A Simple Implementation

- ADD and SUB
  - addU rd, rs, rt
  - subU rd, rs, rt

31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- OR Immediate:
  - ori rt, rs, imm16

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

- LOAD and STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16

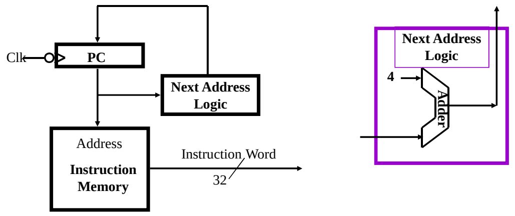
31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

- BRANCH:
  - beq rs, rt, imm16

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

Fetch Instructions

- Fetch instruction, then update PC
- PC updated (at the end of) every cycle
  - What if no branches or jumps?

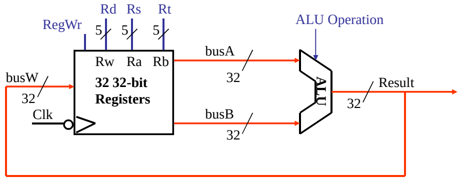


ALU Instructions

- R[rd] <- R[rs] op R[rt] Example: addU rd, rs, rt
  - Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
  - ALUOperation and RegWr: control logic after decoding the instruction

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



Logical operation with Immediate

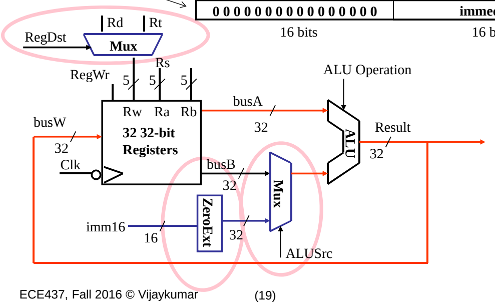
- R[rt] <- R[rs] op ZeroExt[imm16]]
- rd?
- zeroext

31 26 21 16 11 0

op	rs	rt	immediate
6 bits	5 bits	5 bits rd?	16 bits

31 16 15 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	immediate
16 bits	16 bits

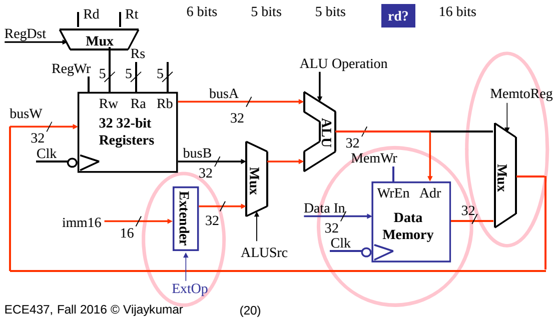


Load Instruction

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: lw rt, imm16(rs)

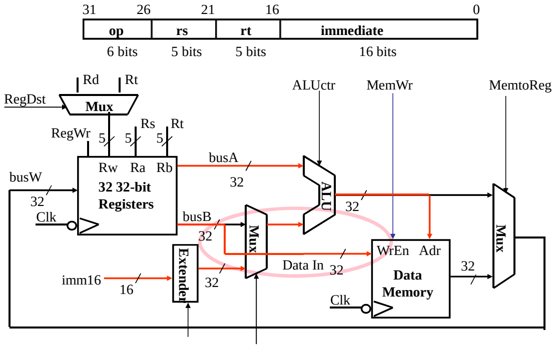
31 26 21 16 11 0

op	rs	rt	immediate
6 bits	5 bits	5 bits rd?	16 bits



Store Instruction

- Mem[R[rs] + SignExt[imm16] <- R[rt]] Example: sw rt, imm16(rs)



Memory

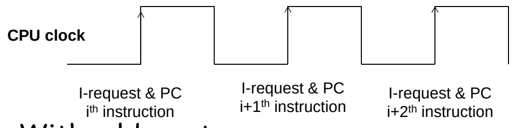
- Used for instruction fetch and data access ld/st
- Memory is big and slow
  - takes 300 CPU clocks today! (ch5)
- Memory can allow only one of instruction fetch or ld/st data access at any given time → need to arbitrate between I-fetch and D-access
  - We will alleviate this in ch5 but will still need arbitration

Memory

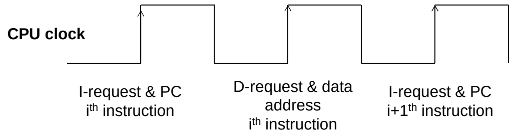
- While one could do both I fetch and D-access one after the other in one (slow) CPU clock, our specific FPGA memory needs two clocks –one each for I-fetch and D-access
  - So our lab design is "mostly single-cycle CPU": one cycle for all instructions except ld and st, and two cycles for ld/st
  - Still you need arbitration so memory does either I-fetch or D-access at a time

Lab memory timing

- No ld/st among instructions



- With a ld or st



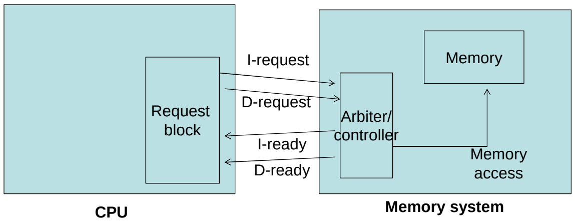
Memory arbitration

- How is this timing realized? Via request-ready handshake + arbitration
- The CPU has a hardware block called “request block” where the CPU generates I-request and D-request signals – non ld/st instructions generate only I-request and ld/st generate both I-request (for I-fetch) and D-request (for ld/st of data)
- The requests go to the “Arbiter block”

Memory arbitration

- The arbiter block chooses one of I-request or D-request if both are active (ld/st) OR I-request if D-request is not active (non-ld/st)
- If both active,
  - the D-request is for current instruction and I-request is for the next instruction
  - D-request gets priority to complete the current instruction before going to next
- The chosen request accesses memory

CPU-Memory Interface



Request block

I-request

D-request

I-ready

D-ready

Arbiter/ controller

Memory

Memory access

CPU

Memory system

- Request is in CPU and arbiter is in memory system -- SEPARATE
- Do not break this interface by going around it – else you will suffer later

Memory arbitration

- When access complete (I or D), the arbiter/controller asserts the corresponding ready (I-ready or D-ready)
  - This is how CPU knows access is complete – this is important in 3 slides
- VERY IMPORTANT: once a D-access is complete, de-assert the D-request
  - Else arbiter will keep choosing that request

Memory arbitration – Detail 1

- For ld/st, the instruction is latched within memory system WHILE data access occurs (else instruction would vanish while data access occurs!)
  - During the middle clock of previous timing diagram
  - This latching is done for you in the code given to you

Memory arbitration- Detail 2

- Lab has two clocks – CPU clock and RAM clock
  - CPU clock is the main clock
  - RAM clock an internal detail to be ignored
  - RAM clock happens to be 2x CPU clock
    - Does not match reality
    - Done to make our specific FPGA work for mostly single-cycle CPU
    - Fixed in the next slide

Memory arbitration

- In lab, memory is variable latency – so your design should work at different latencies (important in real world)
  - Mostly single-cycle is at the lowest latency
  - Longer latencies → multiple clocks per I-fetch or D-access
  - You MAY NOT assume memory will complete within one cycle → you MUST wait for I-ready or D-ready
  - I-ready, D-ready in one cycle at lowest latency & in more cycles at longer latencies

Why?

- Variable latency checks if design is timing-independenct – important concept
- Single memory, so we don't change from two memories (in single cycle) to one memory (in pipelining) – easier for you
- GENERAL- you should test and debug every block BOTH separately AND after merging with rest of design BEFORE going to the next block else you will have a mess that can't be debugged

ORI

- ORI is not in the book
- ORI shows that some instructions need zero extension instead of sign extension
  - Logical vs. arithmetic ops
- What will change in the datapath if ORI is absent?

Conditional Branch Instruction

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- beq rs, rt, imm16
  - IR = Mem[PC] // Fetch the instruction from memory
  - Equal <- R[rs] == R[rt] // Calculate the branch condition
  - if (Equal == 1) // Calculate the next instruction's address
    - PC <- PC + 4 + (SignExt(imm16) << 2)
  - else
    - PC <- PC + 4

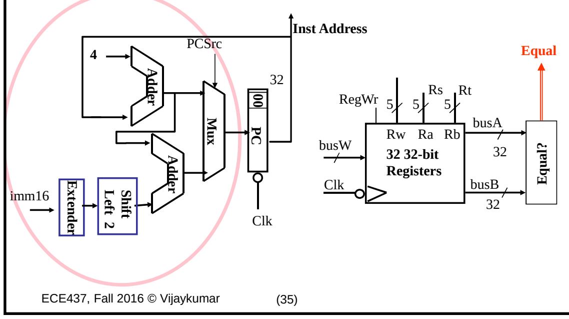
What is this?

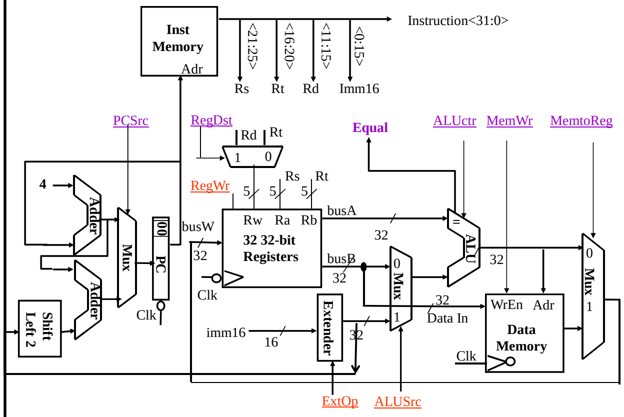
- Branches compute TWO things: branch condition and branch target

Datapath for 'beq'

- beq rs, rt, imm16
  - Datapath generates branch condition (Equal)

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits





Summary

- For a given instruction
  - Describe operation in RTL
  - Use ALUs, Registers, Memory, adders to achieve reqd. functionality
- To add instructions
  - Rinse and repeat; Reuse components via muxes
  - Not all blocks are used by all instrs so you need to ensure unused blocks don't interfere
- Controls : next
  - Selection controls for muxes
  - ALU controls for ALU ops
  - Register address controls
  - Write enables for registers/memory

Exercise

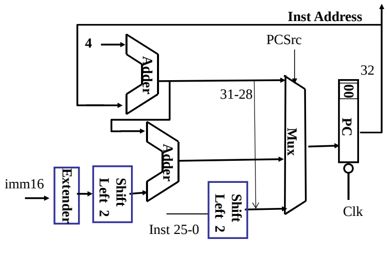
- Add jump instruction to single cycle datapath
  - j Addr
  - RTL
    - PC <- (PC+4)[31:28] // Addr // 00

J-type	op	target address	jump

- See worksheet #1

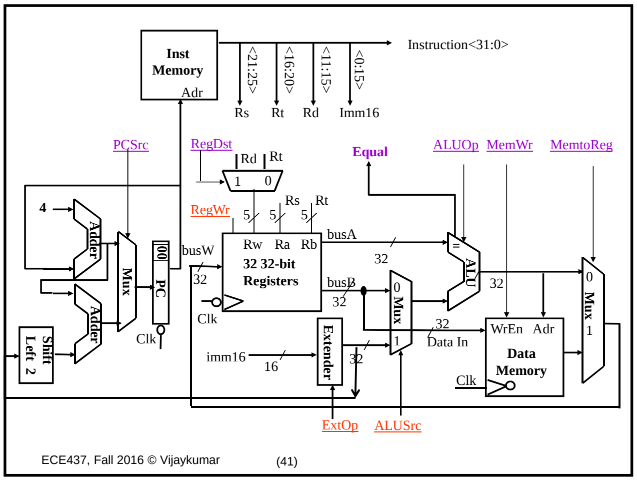
Exercise

J-type	op	target address	jump

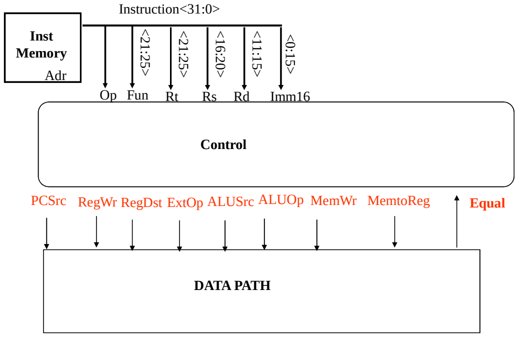


Exercise

- See worksheet (Fig 4.15)
- Highlight active datapath for
  - Add
  - Beq
  - Sw
  - Lw

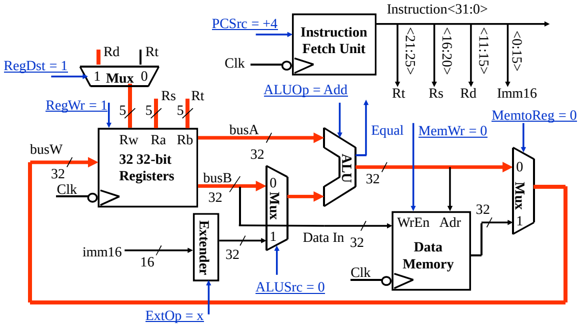


Control for Datapath



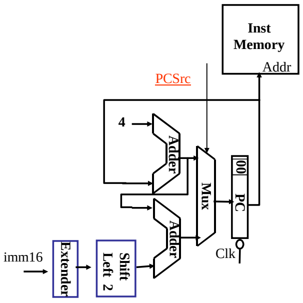
Controls for Add Operation

- R[rd] = R[rs] + R[rt]



Meaning of Control Signals

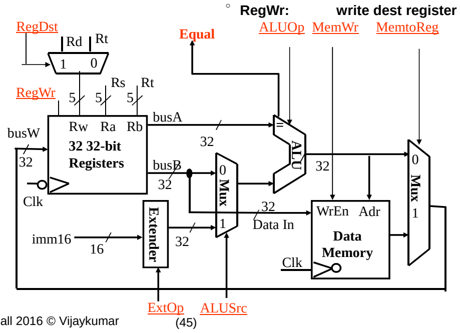
- rs, rt, rd and imm16 hardwired in datapath
- PCSrc: 0 => PC <- PC + 4; 1 => PC <- PC + 4 + SignExt(Im16) || 00



Meaning of Control Signals

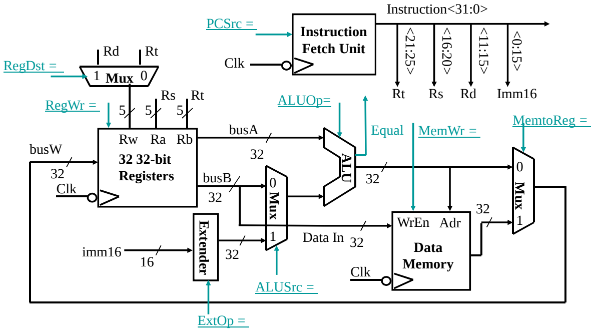
- ExtOp: "zeroext", "signext"
- ALUsrc: 0 => regB; 1 => immed
- ALUOp: "add", "sub", "or"

- MemWr: write memory
- MemtoReg: 1 => Mem
- RegDst: 0 => "rt"; 1 => "rd"
- RegWr: write dest register



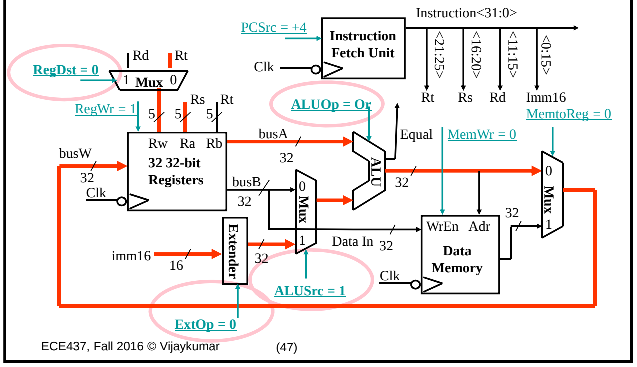
ORI Controls: Worksheet

- R[rt] <- R[rs] or ZeroExt[Imm16]



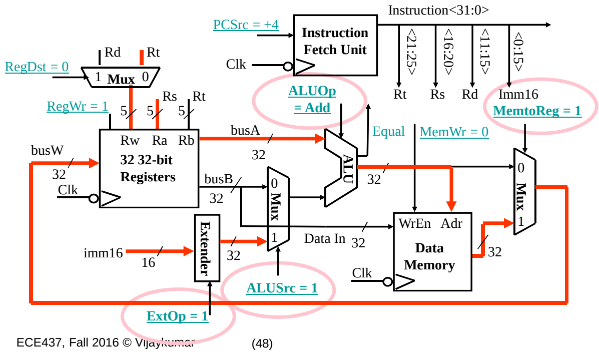
ORI Controls: Solution

- R[rt] <- R[rs] or ZeroExt[Imm16]



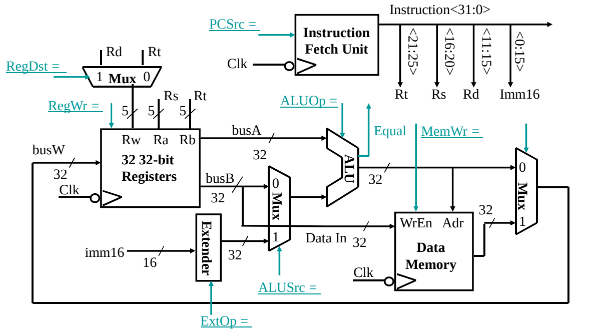
LW Controls

- R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



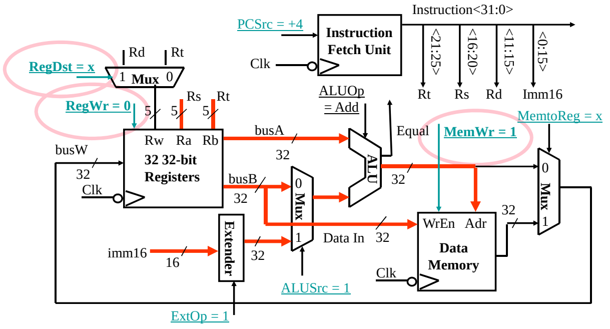
SW Controls: Worksheet

- R[rt] -> Data Memory {R[rs] + SignExt[imm16]}



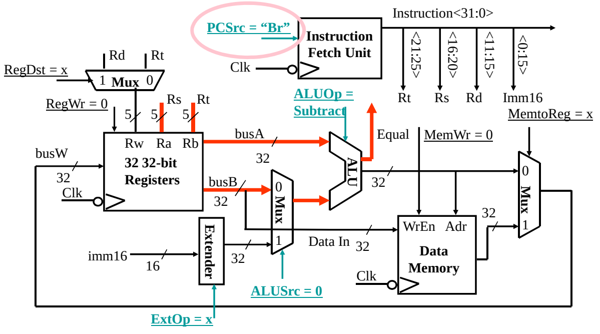
SW Controls: Solution

- R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



BEQ Controls

- if (R[rs] - R[rt] == 0) then Equal <- 1 ; else Equal <- 0



Summary of Control Signals

inst **Register Transfer**

ADD R[rd] <- R[rs] + R[rt]; PC <- PC + 4
ALUsrc = RegB, ALUOp = "add", RegDst = rd, RegWr, PCSrc = "+4"

SUB R[rd] <- R[rs] – R[rt]; PC <- PC + 4
ALUsrc = RegB, ALUOp = "sub", RegDst = rd, RegWr, PCSrc = "+4"

ORi R[rt] <- R[rs] + zero_ext(Imm16); PC <- PC + 4
ALUsrc = Im, Extop = "Z", ALUOp = "or", RegDst = rt, RegWr, PCSrc = "+4"

LOAD R[rt] <- MEM[R[rs] + sign_ext(Imm16)]; PC <- PC + 4
ALUsrc = Im, Extop = "Sn", ALUOp = "add",
MemtoReg, RegDst = rt, RegWr, PCSrc = "+4"

STORE MEM[R[rs] + sign_ext(Imm16)] <- R[rs]; PC <- PC + 4
ALUsrc = Im, Extop = "Sn", ALUOp = "add", MemWr, PCSrc = "+4"

BEQ if (R[rs] == R[rt]) then PC <- PC + sign_ext(Imm16)] || 00 else PC <- PC + 4;
ALUsrc = RegB, PCSrc = "Beq AND Equal", ALUOp = "sub"

Control Logic

- Logic must generate appropriate signals for all instructions
- Summary slide (previous)
  - A way of representing the truth table
- Till now: Instr → signal, next: transpose
  - First: Equations in terms of opcodes
  - Next: Equations in terms of instruction bits

Controls: Logic equations

- PCSrc <= if (OP == BEQ) then Equal else 0
- ALUsrc <=if (OP == "R-type") then "regB"
  elseif (OP == BEQ) then regB, else "imm"
- ALUOp <= if (OP == "R-type") then **funct**
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp <= ____________
- MemWr <= ____________
- MemtoReg <= ____________
- RegWr: <= ____________
- RegDst: <= ____________

Controls: Logic equations

- PCSrc <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "R-type") then "regB"
  elseif (OP == BEQ) then regB, else "imm"
- ALUOp <= if (OP == "R-type") then **funct**
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp <= if (OP == ORi) then "zeroext" else "signext"
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

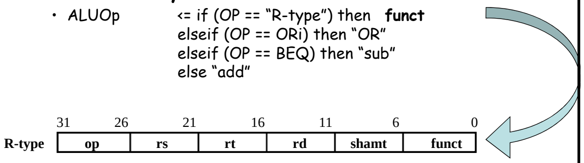
Truth Table summary

See Appendix B → **func**	10 0000	10 0010	**We Don't Care :-)**				
→ **op**	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
PCSrc	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	1	x
ALUOp<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

	31	26	21	16	11	6	0	
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**		**add, sub**
I-type	**op**	**rs**	**rt**	**immediate**				**ori, lw, sw, beq**
J-type	**op**	**target address**						**jump**

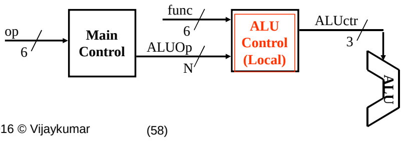
Local vs Global Control

- One more layer of abstraction
  - ALUOp <= if (OP == "R-type") then **funct**
    elseif (OP == ORi) then "OR"
    elseif (OP == BEQ) then "sub"
    else "add"

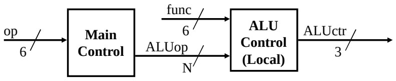


Global Control: Truth Table

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUOp<N-1:0>	"R-type"	Or	Add	Add	Subtract	xxx



Encoding

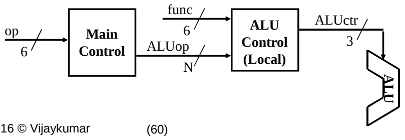


- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

Global Control: Truth Table

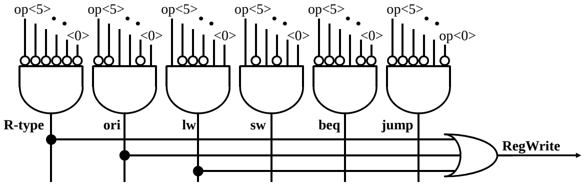
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<2:0>	"R-type"	Or	Add	Add	Subtract	xxx
	(100)	(010)	(000)	(000)	(001)	



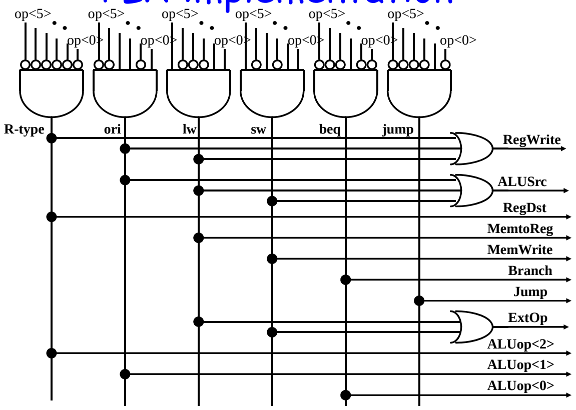
Truth Table for RegWrite

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

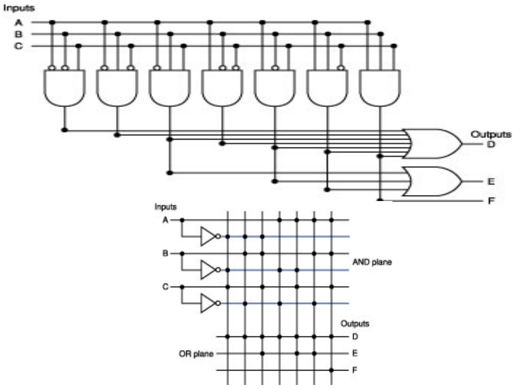
- RegWrite = R-type + ori + lw
  - = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  - + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  - + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



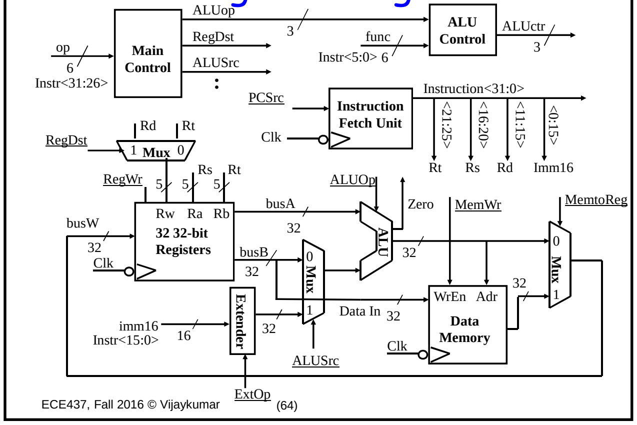
PLA implementation

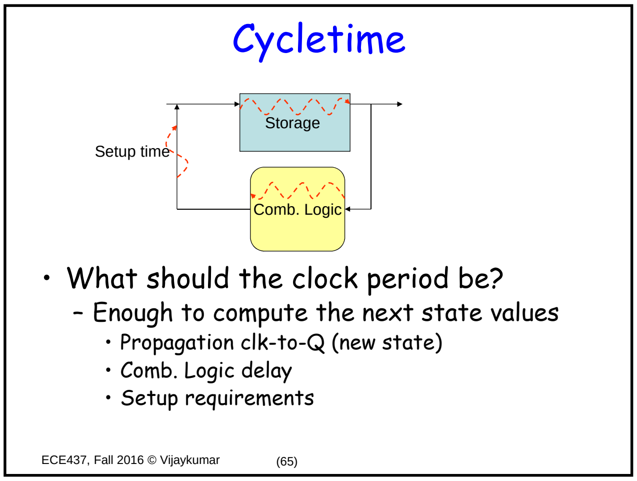


PLA Representation



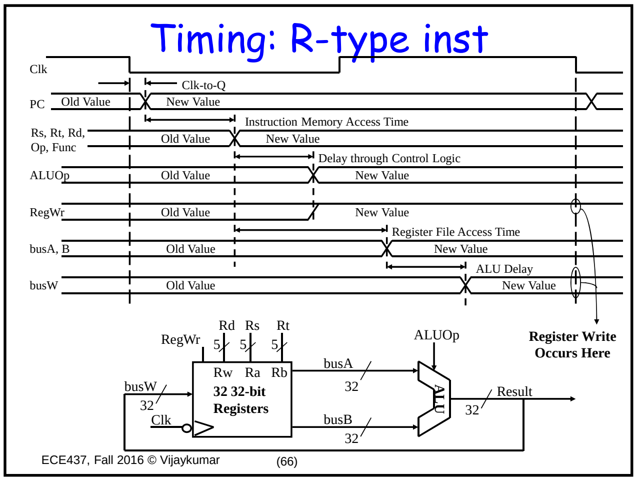
Putting it all together



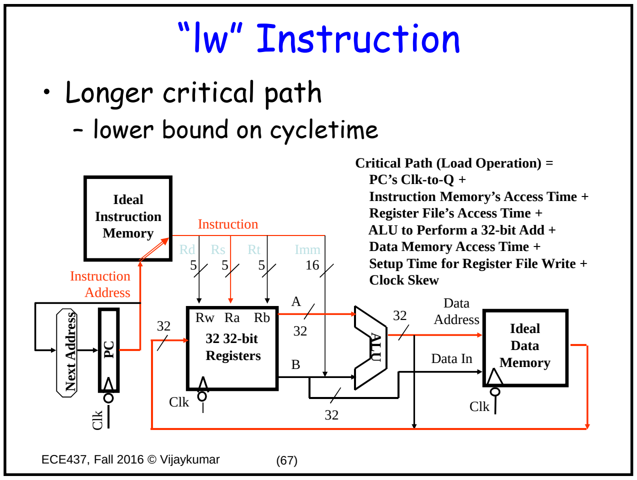


Cycletime

- What should the clock period be?
  - Enough to compute the next state values
    - Propagation clk-to-Q (new state)
    - Comb. Logic delay
    - Setup requirements



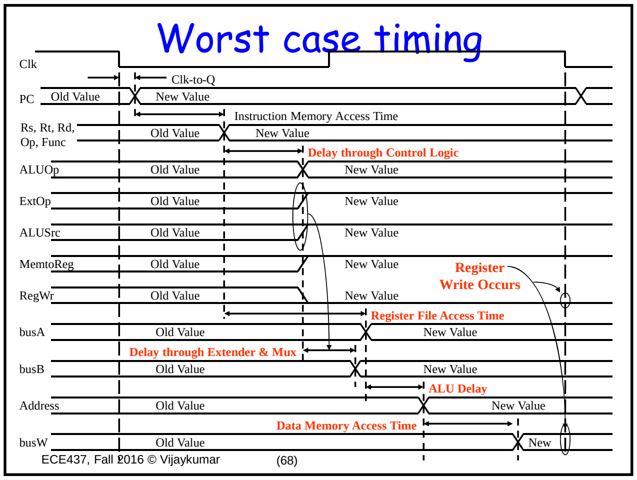
Timing: R-type inst



"lw" Instruction

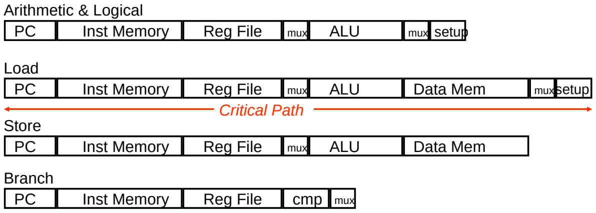
- Longer critical path
  - lower bound on cycletime

Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew



Worst case timing

What's wrong with our processor?


Arithmetic & Logical
PC | Inst Memory | Reg File | mux | ALU | mux | setup

Load
PC | Inst Memory | Reg File | mux | ALU | Data Mem | mux | setup
Critical Path

Store
PC | Inst Memory | Reg File | mux | ALU | Data Mem

Branch
PC | Inst Memory | Reg File | cmp | mux

- LOOOOONG Cycle Time
- ALL instructions take as much time as the slowest
- Real memory MUCH slower than our idealized memory
  - Today some 100ns (memory+bus+control) vs. 0.25ns CPU clock
  - cannot finish in one (short) cycle

Notion of Performance

- We need to understand "performance" better because in the rest of course we will improve the performance of our processor
- To understand performance we will go to chapter 1b (1.4 onwards) before returning to chapter 4b (4.5 onwards)