U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

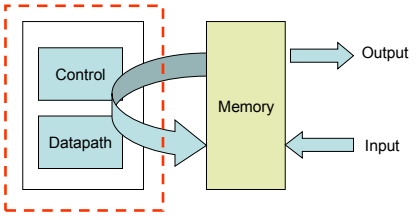
Prof. Karu Sankaralingam

Single-Cycle Processor (5.1-5.4)

www.cs.wisc.edu/~karu/courses/cs552

Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar,Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Processor Implementation



Outline

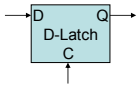
- Sequential logic & Clocking methodology
- Single-Cycle Datapath - 1 CPI
- Single-Cycle Control
- Defer to Later
  - Multiple cycle implementation
  - Microprogramming
  - Exceptions

Review Sequential Logic

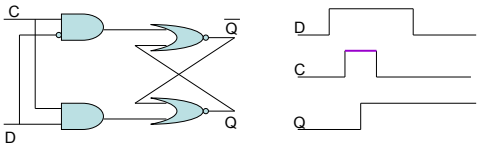
- Logic is combinational if output is solely function of inputs
  - e.g., ALU
- Logic is sequential or "has state" if output is a function of
  - past and current inputs
  - past inputs "remembered" in "state"
  - but no magic!

Review: D Latch

Interface →

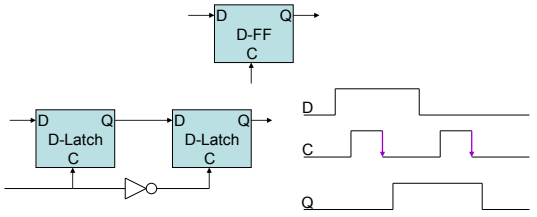


Implementation:



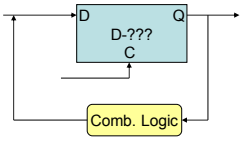
- Clock high: Q = D, after some min to max propagate delay
- Clock low: Q, Q remain unchanged
- Sensitive to clock level

Review: D Flip-flop



- D flip-flop - built from 2 D-latches
  - while clock high, D flows into 1st latch, but not 2nd
  - in 2nd Q retains old value
- Remember D at falling edge & propagate thru 2nd latch

Review Latch vs. Flip-Flop

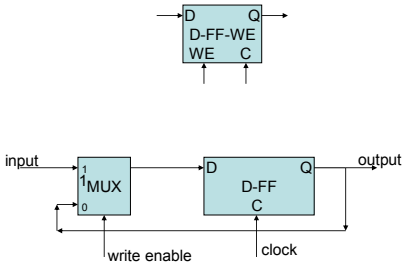


- Can build with flip-flops.
- Why does this fail for latch?
  - Q may rush via feedback path back to D before clock falls

D-FF WriteEnable (forbidden design)

- What if D-FF state unchanged on some clock cycles?
- Logically add "WriteEnable"
- Implementation could AND Clock & WriteEnable
  - Called "clock gating" or "qualifying the clock"
  - Forbidden (in this class)
- Real designs do clock gating but tricky due to glitches

D-FF WriteEnable (preferred design)

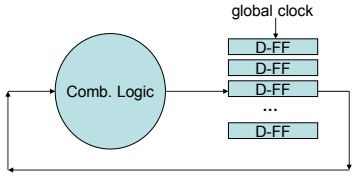


552 Clocking Methodology Goals

- Simplicity → Correctness
- Restricts freedom but eliminates errors
- Allows design datapath/control without thinking about clocks

552 Clocking Methodology Rules

- We provide D-FF design
- Use this D-FF for all processor state
- Same unqualified clock for all D-FFs
- Combinational logic must finish in one cycle

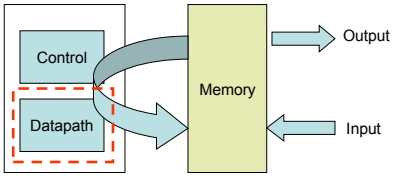


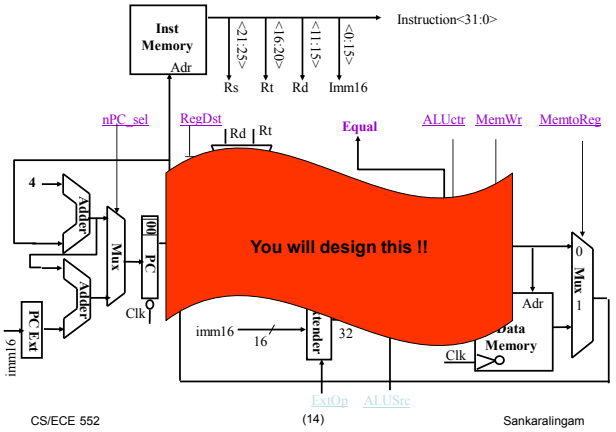
552 Clocking Methodology Implications

- Clock Becomes Implicit
  - same clock; same edge; no glitches
- You concentrate on getting logic correct
- Clock cycle time determined by worst-case sequential logic delay (plus a little)
- When you're paid the big bucks, …

Processor Implementation

- Next : Single-Cycle Datapath



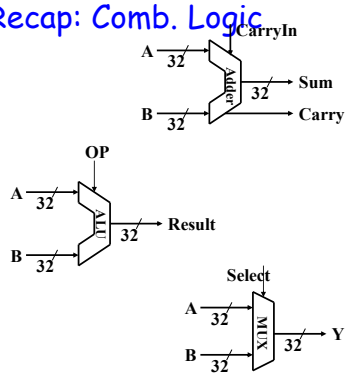


Outline

- Sequential logic design review
- Clocking methodology
- Datapath - 1 CPI
  - single instruction, 2's complement, unsigned
- Control
- Multiple cycle implementation
- Microprogramming
- Exceptions

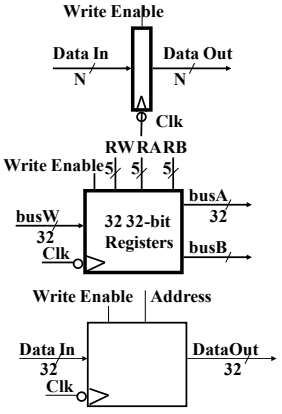
Recap: Comb. Logic

- Adder
- ALU
- Mux

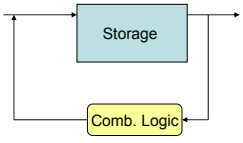


Recap:Storage

- Register
  - for PC
- Register file
  - 32 registers
  - 2 read ports/buses
  - 1 write port/bus
- Memory
  - 1 input bus
  - 1 output bus
  - Not bidirectional

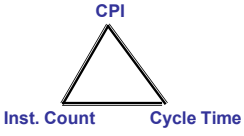


Computer as State Machine



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

Processor Implementation



- Implementation determines
  - CPI
  - Cycle time
- Inst. Count = f(interface,compiler)
  - Interface = ISA, endianness etc.

Datapath - 1 CPI

- Assumption: Get one whole instruction done in one long cycle
  - fetch, decode/read operands, execute, memory, writeback
  - useful way to represent steps and identify required datapath elements: RTL
- For single instruction
- Put it together

Register Transfer Language

- RTL gives the meaning of the instructions
- All start by fetching the instruction

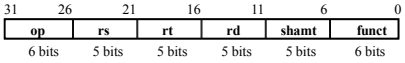
op | rs | rt | rd | shamt | funct = MEM[PC]

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

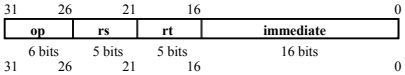
inst	Register Transfers	
ADDU	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUBU	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rt];	PC <– PC + 4
BEQ	if (R[rs] == R[rt]) then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

A Simple Implementation

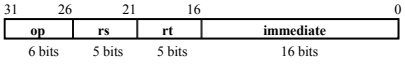
- ADD and SUB
  - addU rd, rs, rt
  - subU rd, rs, rt



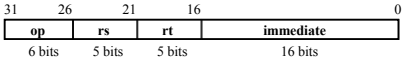
- OR Immediate:
  - ori rt, rs, imm16



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

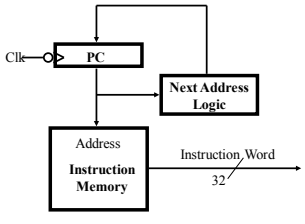


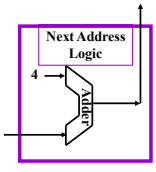
- BRANCH:
  - beq rs, rt, imm16



Fetch Instructions

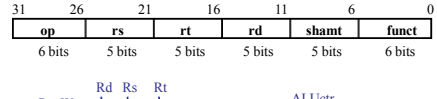
- Fetch instruction, then update PC
- PC updated (at the end of) every cycle
  - What if no branches or jumps?

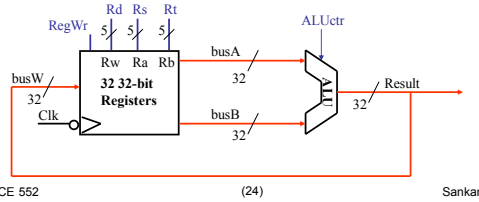




ALU Instructions

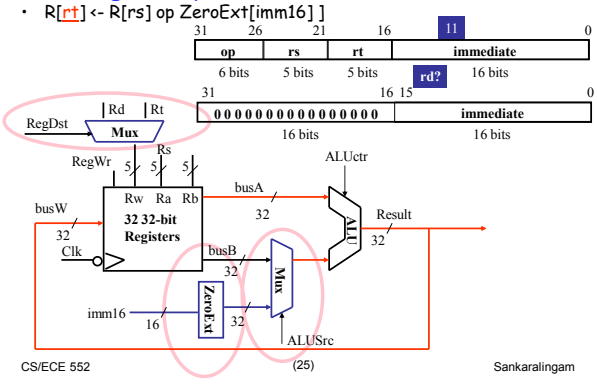
- R[rd] <- R[rs] op R[rt] Example: addU rd, rs, rt
  - Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
  - ALUctr and RegWr: control logic after decoding the instruction





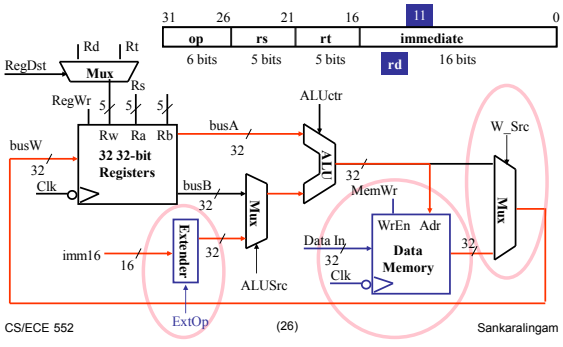
Logical operation with Immediate

- R[rt] <- R[rs] op ZeroExt[imm16]]



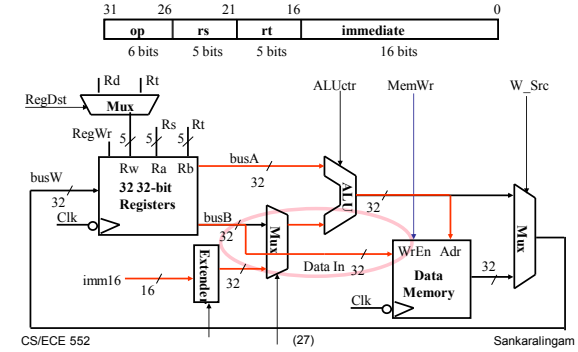
Load Instruction

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

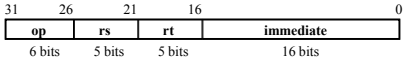


Store Instruction

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



Conditional Branch Instruction

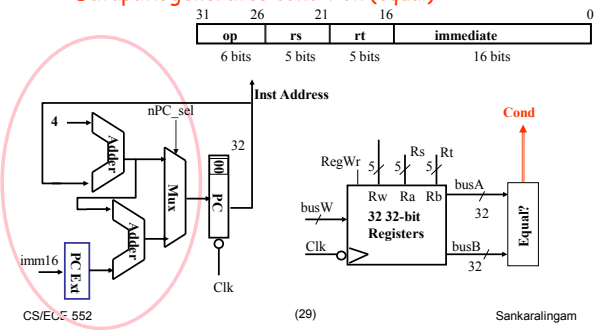


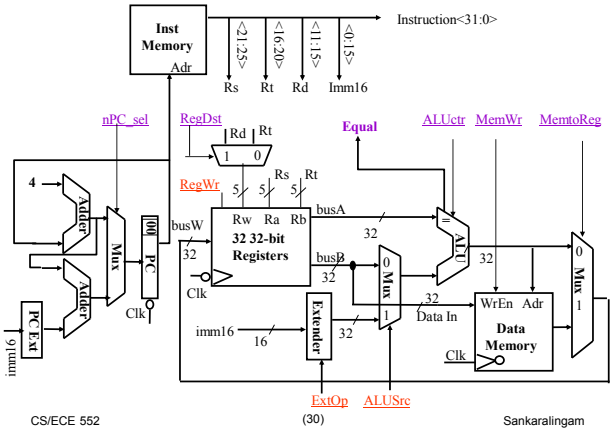
- beq rs, rt, imm16
  - IR = Mem[PC] // Fetch the instruction from memory
  - Equal <- R[rs] == R[rt] // Calculate the branch condition
  - if (COND eq 0) // Calculate the next instruction's address
    - PC <- PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC <- PC + 4

What is this?

Datapath for 'beq'

- beq rs, rt, imm16
  - Datapath generates condition (equal)

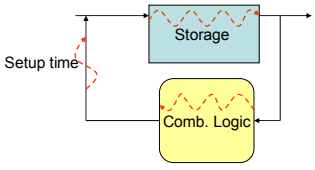




Summary

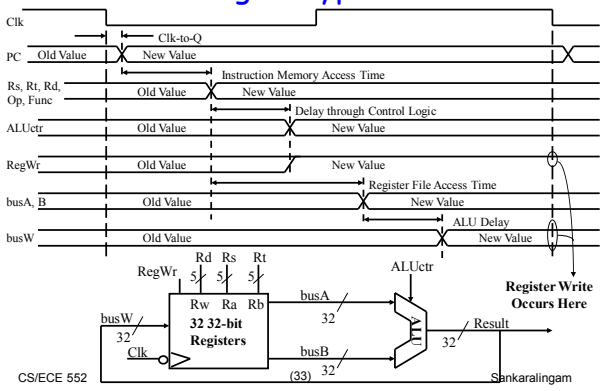
- For a given instruction
  - Describe operation in RTL
  - Use ALUs, Registers, Memory, adders to achieve reqd. functionality
- To add instructions
  - Rinse and repeat
  - Reuse components by using muxes
- Controls : later
  - Selection controls for muxes
  - ALU controls for ALU ops
  - Register address controls
  - Write enables for registers/memory

Cycletime



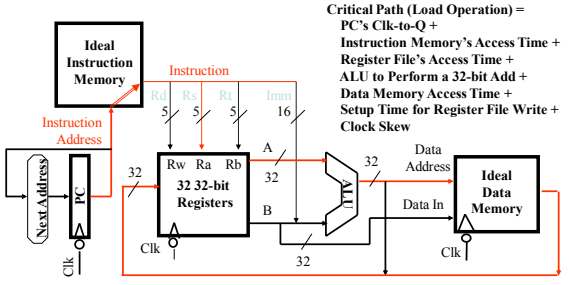
- What should the clock period be?
  - Enough to compute the next state values
    - Propagation clk-to-Q (new state)
    - Comb. Logic delay
    - Setup requirements

Timing: R-type inst

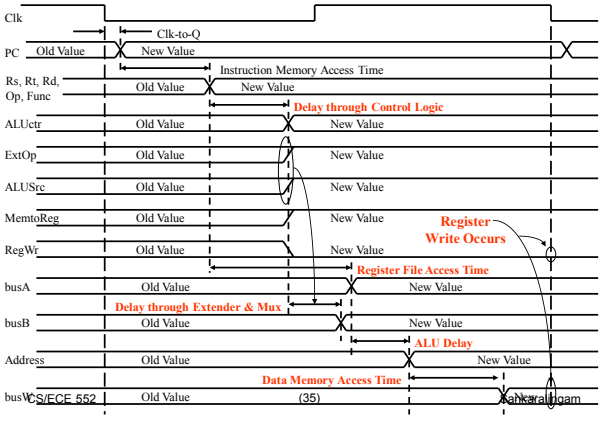


"lw" Instruction

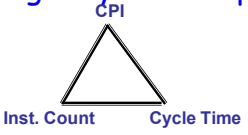
- Longer critical path
  - lower bound on cycletime



Worst case timing



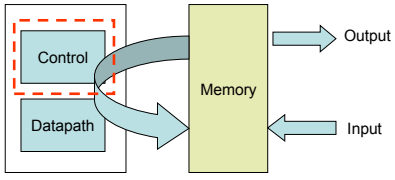
Single-cycle Datapath



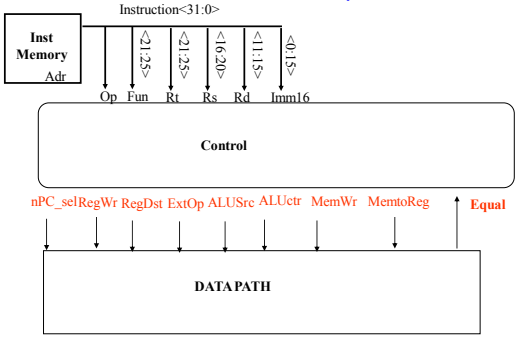
- Performance Implications
  - Minimize all three
  - Insts/prog fixed -- f(interface,compiler)
  - CPI = 1 : As good as it gets (*)
  - Clock cycle time : high, "lw" critical path

Processor Implementation

- Next : Control for Single-Cycle Datapath

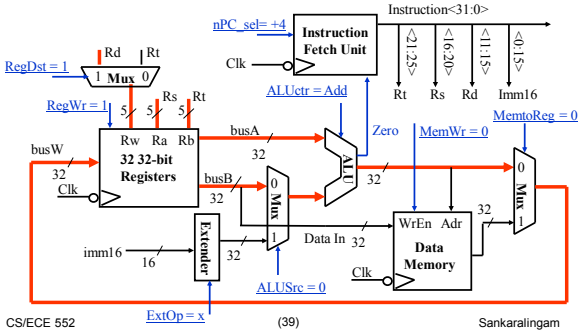


Control for Datapath



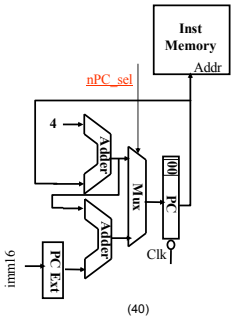
Controls for Add Operation

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



Meaning of Control Signals

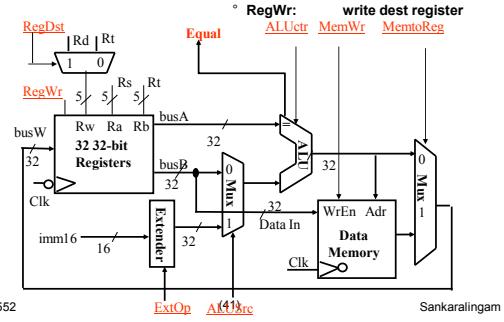
- rs, rt, rd and imm16 hardwired in datapath
- nPC_sel: 0 => PC <- PC + 4; 1 => PC <- PC + 4 + SignExt(Im16) || 00



Meaning of Control Signals

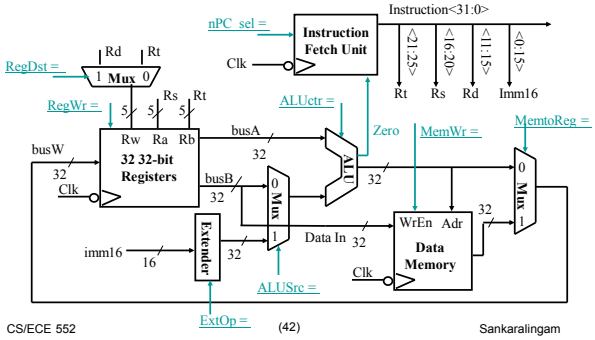
- ExtOp: "zero", "sign"
- ALUsrc: 0 => regB; 1 => immed
- ALUctr: "add", "sub", "or"

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



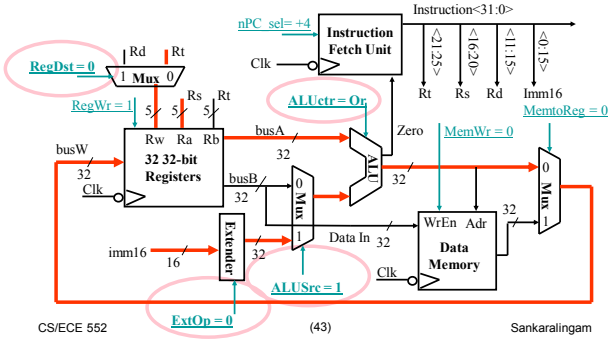
ORI Controls: Worksheet

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



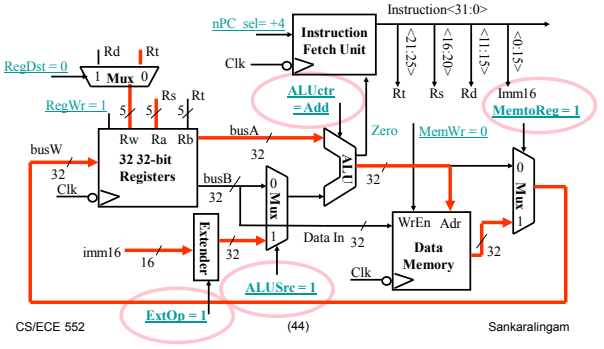
ORI Controls: Solution

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



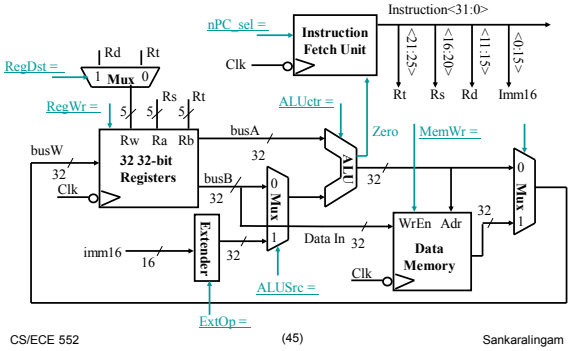
LW Controls

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



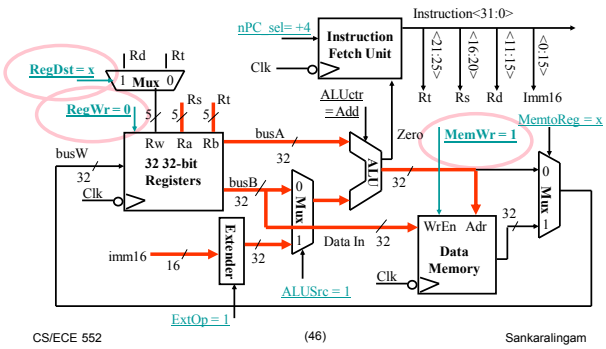
SW Controls: Worksheet

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



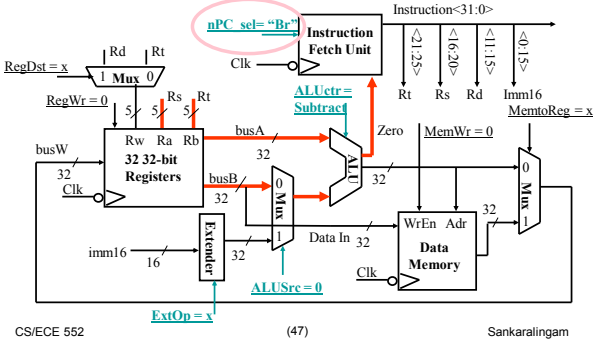
SW Controls: Worksheet

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



BEQ Controls

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



Summary of Control Signals

inst **Register Transfer**

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

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

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

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

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

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

Control Logic

- Logic must generate appropriate signals for all instructions
- Summary slide (previous)
  - A way of representing the truth table
- First:
  - Equations in terms of opcodes
- Next
  - Equations in terms of instruction bits

Controls: Logic equations

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <=if (OP == "R-type") then "regB"
  elseif (OP == BEQ) then regB, else "imm"
- ALUctr <= if (OP == "R-type") then **funct**
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp <= _____
- MemWr <= _____
- MemtoReg <= _____
- RegWr: <= _____
- RegDst: <= _____

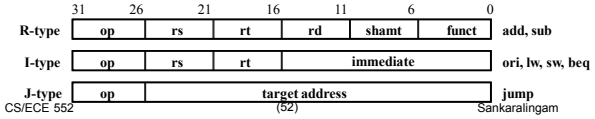
Controls: Logic equations

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "R-type") then "regB"
  elseif (OP == BEQ) then regB, else "imm"
- ALUctr <= if (OP == "R-type") then **funct**
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub" else "add"
- ExtOp <= if (OP == ORi) then "zero" else "sign"
- 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

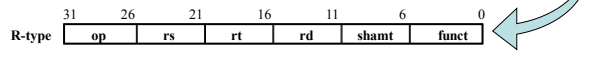
See Appendix A

	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
nPCsel		0	0	0	0	0	1	0
Jump		0	0	0	0	0	0	1
ExtOp		x	x	0	1	1	x	x
ALUctr<2:0>		Add	Subtract	Or	Add	Add	Subtract	xxx



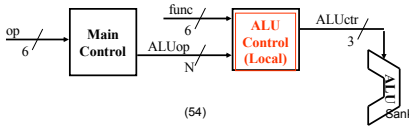
Local vs Global Control

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

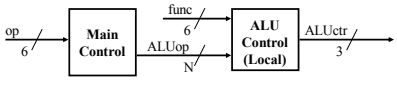


Global Control: Truth Table

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:0>	"R-type"	Or	Add	Add	Subtract	xxx



Encoding

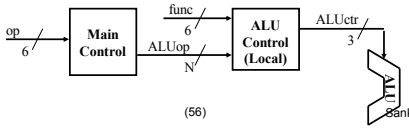


- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type"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

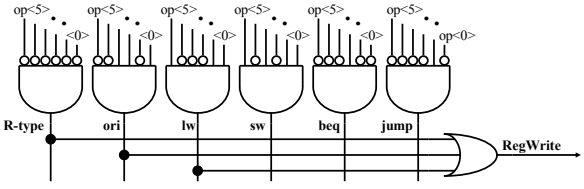
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<2:0>	"R-type" (100)	Or (010)	Add (000)	Add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



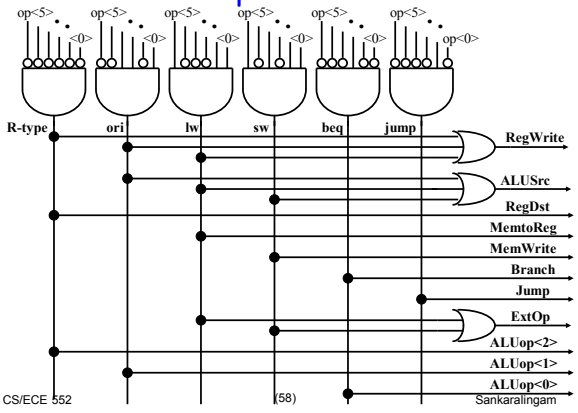
Truth Table for RegWrite

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

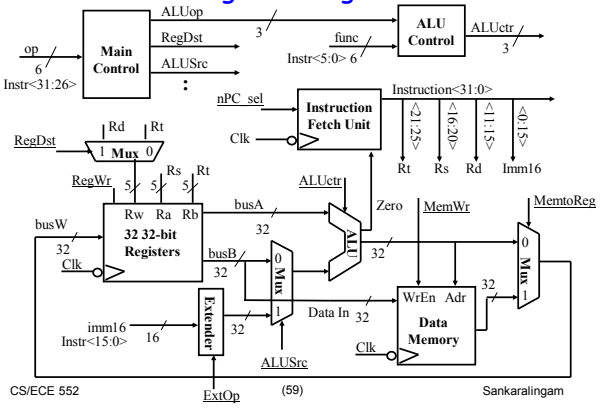
- RegWrite = R-type + ori + lw
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



PLA implementation



Putting it all together



Outline

- Sequential logic & Clocking methodology
- Single-Cycle Datapath - 1 CPI
- Single-Cycle Control
- Defer to Later
  - Multiple cycle implementation
  - Microprogramming
  - Exceptions