CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Single Cycle MIPS CPU

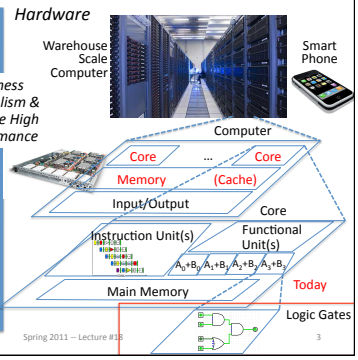
Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp11

You Are Here!

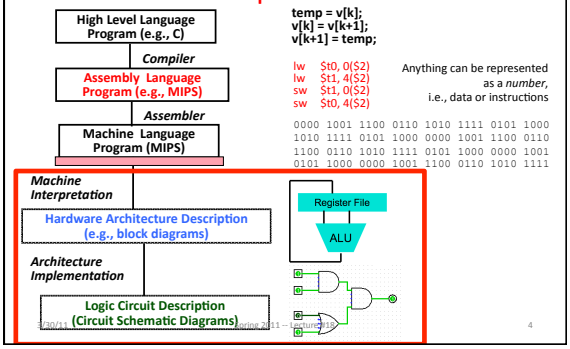
Software / *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates functioning in parallel at same time

Harness Parallelism & Achieve High Performance



Levels of Representation/ Interpretation



Review

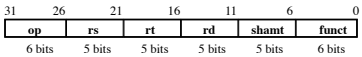
- Clocks tell us when D-flip-flops change
  - Setup and Hold times important
- We pipeline long-delay CL for faster clock
- Finite State Machines extremely useful
- Use muxes to select among input
  - S input bits selects 2^S inputs
  - Each input can be n-bits wide, indep of S
- Can implement muxes hierarchically
- Can implement FSM with register + logic

Agenda

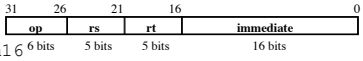
- MIPS-lite Datapath
- Administrivia
- CPU Timing
- MIPS-lite Control
- Datapath Control
- Technology Break
- Control Implementation

The MIPS-lite Subset

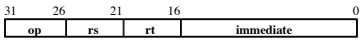
- ADDU and SUBU
  - `addu rd,rs,rt`
  - `subu rd,rs,rt`



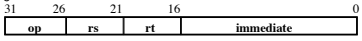
- OR Immediate:
  - `ori rt,rs,imm16`



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



- BRANCH:
  - `beq rs,rt,imm16`



Processor Design Process

- Five steps to design a processor:
  - Step 1: Analyze instruction set to determine datapath requirements (see next slide)
  - Step 2: Select set of datapath components & establish clocking methodology
  - Step 3: Assemble datapath components that meet the requirements
  - Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer
  - Step 5: Assemble the control logic

Register Transfer Language (RTL)

- RTL gives the meaning of the instructions

```
{op , rs , rt , rd , shamt , funct} ← MEM[ PC ]
{op , rs , rt ,   Imm16} ← MEM[ PC ]
```

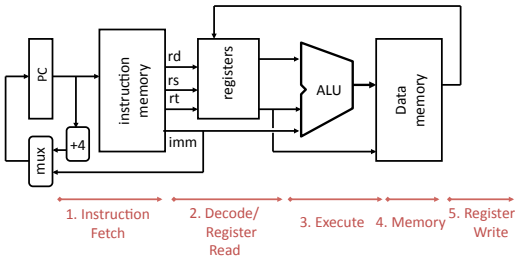
- All start by fetching the instruction

```
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 + 4 + (sign_ext(Imm16) || 00)
           else PC ← PC + 4
```


Step 1: Requirements of the Instruction Set

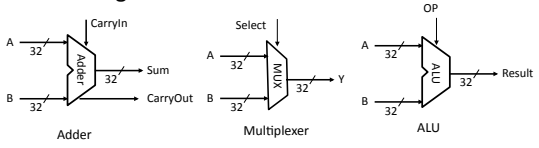
- Memory (MEM)
  - Instructions & data (will use one for each: really caches)
- Registers (R: 32 x 32)
  - Read *rs*
  - Read *rt*
  - Write *rt* or *rd*
- PC
- Extender (sign/zero extend)
- Add/Sub/OR unit for operation on register(s) or extended immediate
- Add 4 (+ maybe extended immediate) to PC
- Compare if registers equal?

Generic Steps of Datapath



Step 2: Components of the Datapath

- Combinational Elements
- Storage Elements + Clocking Methodology
- Building Blocks



ALU Needs for MIPS-lite + Rest of MIPS

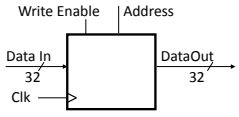
- Addition, subtraction, logical OR, ==:

```
ADDU   R[rd] = R[rs] + R[rt]; ...
SUBU   R[rd] = R[rs] – R[rt]; ...
ORI    R[rt] = R[rs] | zero_ext(Imm16)...
BEQ    if ( R[rs] == R[rt] )...
```

- Test to see if output == 0 for any ALU operation gives == test. How?
- P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)
- ALU from Appendix C, section C.5

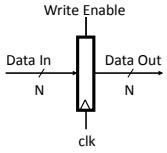
Storage Element: Idealized Memory

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid ⇒ Data Out valid after "access time"



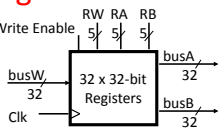
Storage Element: Register (Building Block)

- Similar to D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - Negated (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on rising edge of clock



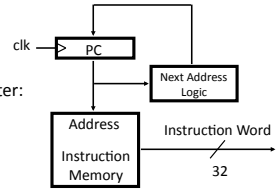
Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after "access time."



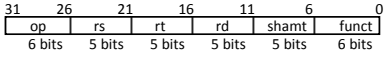
Step 3: Assemble DataPath Meeting Requirements

- Register Transfer Requirements ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation
- Common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC ← PC + 4
    - Branch and Jump: PC ← "something else"

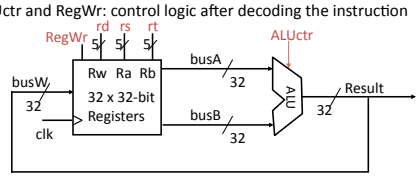


Step 3: Add & Subtract

- `R[rd] = R[rs] op R[rt]` (`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

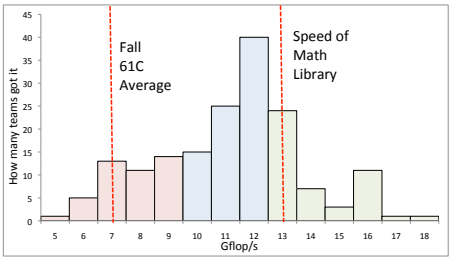


- … Already defined the register file & ALU

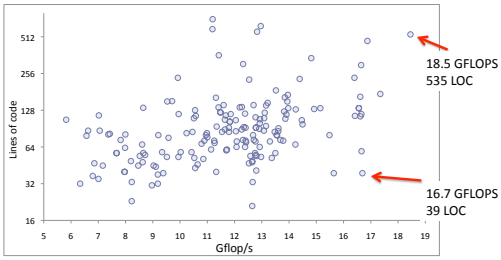
Administrivia

- Project 3, Part 2 due Sunday 4/3
  - Threads Level Parallelism and OpenMP
- Project 4, Part 1 due Sunday 4/10
  - Design a 16-bit pipelined computer in Logisim
  - Last homework due Sunday 4/10
  - Project 4, Part 2 due Sunday 4/17
- Extra Credit due 4/24 – Fastest Matrix Multiply
- Final Exam Monday 5/9 11:30-2:30PM

Project 3 speeds



Lines of Code vs. Performance



Administrivia

- What classes should I take (now)?
- Take classes from great teachers! (teacher > class)
  - Distinguished Teaching Award (very hard to get: ~3/year)
    - http://teaching.berkeley.edu/dta-dept.html
  - HKN Course evaluations (≥6 is very good)
    - https://hkn.eecs.berkeley.edu/coursesurveys
  - EECS web site has plan for year (up in late spring: now)
    - http://www.eecs.berkeley.edu/Scheduling/CS/schedule-draft.html
- If have choice of multiple great teachers
  - EE122 Networking
  - CS152 Computer Architecture and Engineering
  - CS162 Operating Systems and Systems Programming
  - CS169 Software Engineering (for SaaS with Fox)
  - CS194 Engineering Parallel Software (offered in Fall?)

61c in the News

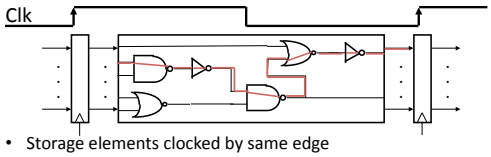
- World's most admired companies
  - Published by *Fortune* magazine, March 2011
- Picked by CEOs of 5000 companies
- IT companies: 3 of top 5, 1/3 top 12 in all industries
  - Client and Cloud focus

1 Apple (mobile client)
2 Google (cloud)
3 Berkshire Hathaway (invest)
4 Johnson & Johnson (health)
5 Amazon.com (cloud)
6 Procter & Gamble (consumer)
7 Toyota Motor (car)
8 Goldman Sachs (finance)
9 Wal-Mart Stores (retail)
10 Coca-Cola (beverage)
11 Microsoft (PC->client-cloud)
12 Southwest Airlines (airline)

Agenda

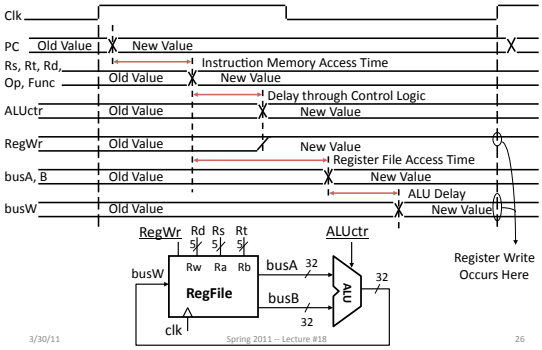
- MIPS-lite Datapath
- Administrivia
- CPU Timing
- MIPS-lite Control
- Datapath Control
- Technology Break
- Control Implementation

Clocking Methodology



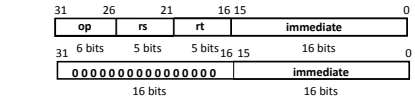
- Storage elements clocked by same edge
- "Critical path" (longest path through logic) determines length of clock period
- Have to allow for Clock to Q and Setup Times too
- This lecture (and P&H sections) 4.3-4.4 do whole instruction in 1 clock cycle for pedagogic reasons
  - Project 4 will do it in 2 clock cycles via simple pipelining
  - Next week explain pipelining and use 5 clock cycles per instruction

Register-Register Timing: One Complete Cycle

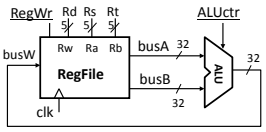


Logical Operations with Immediate

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

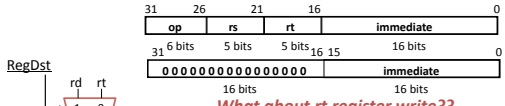


But we're writing to Rt register??

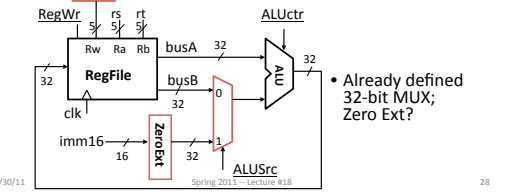


Logical Operations with Immediate

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



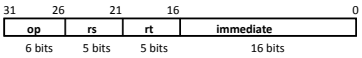
What about rt register write??

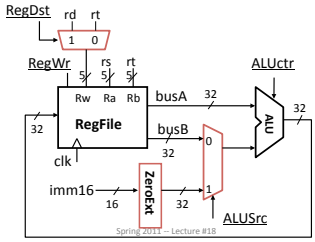


- Already defined 32-bit MUX; Zero Ext?

Load Operations

- R[rt] = Mem[R[rs] + SignExt[imm16]]
  Example: `lw rt,rs,imm16`

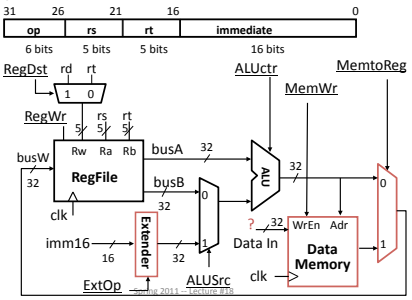




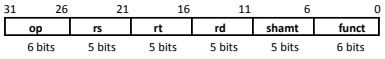
What sign extending??

Load Operations

- R[rt] = Mem[R[rs] + SignExt[imm16]]
  Example: `lw rt,rs,imm16`



RTL: The `Add` Instruction

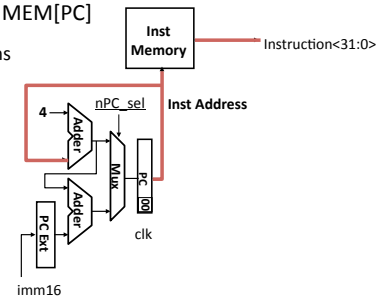


`add rd, rs, rt`

- MEM[PC] Fetch the instruction from memory
- R[rd] = R[rs] + R[rt] The actual operation
- PC = PC + 4 Calculate the next instruction's address

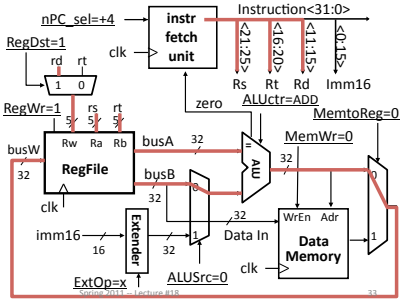
Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory:
  Instruction = MEM[PC]
  - same for all instructions



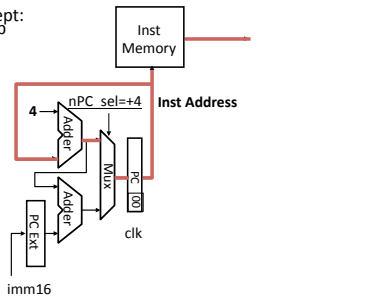
Single Cycle Datapath during Add

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

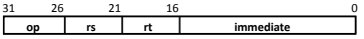


Instruction Fetch Unit at End of Add

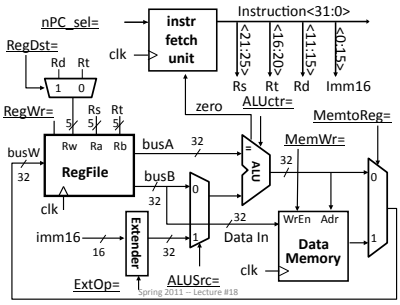
- PC = PC + 4
  - Same for all instructions except: Branch and Jump



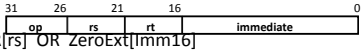
Single Cycle Datapath during Or Immediate



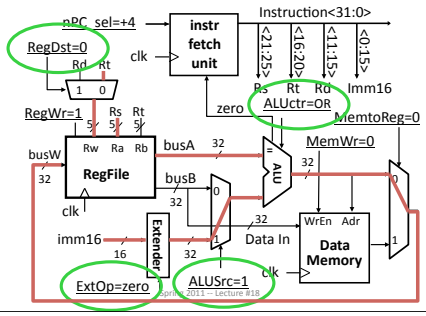
- R[rt] = R[rs] OR ZeroExt[Imm16]



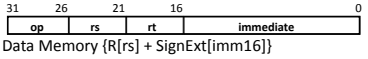
Single Cycle Datapath during Or Immediate



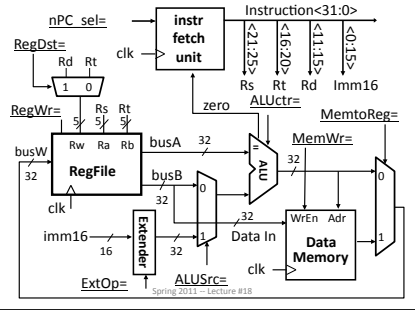
- R[rt] = R[rs] OR ZeroExt[Imm16]



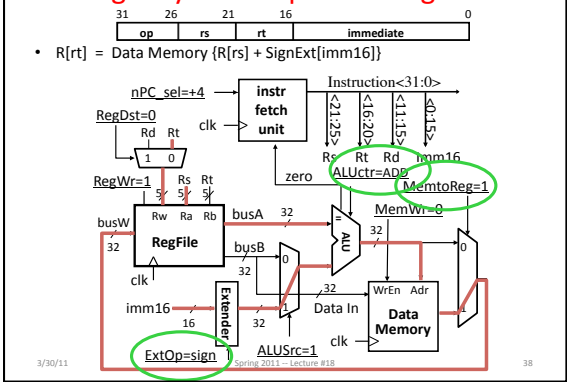
Single Cycle Datapath during Load



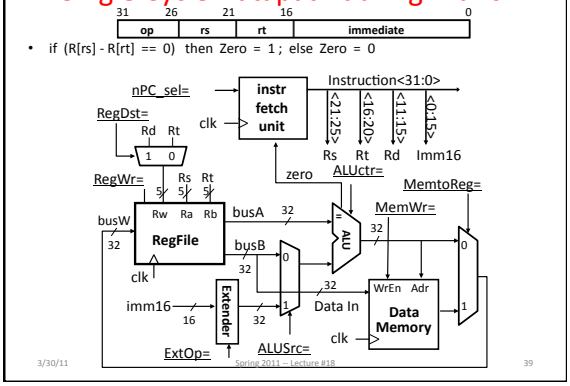
- R[rt] = Data Memory {R[rs] + SignExt[imm16]}



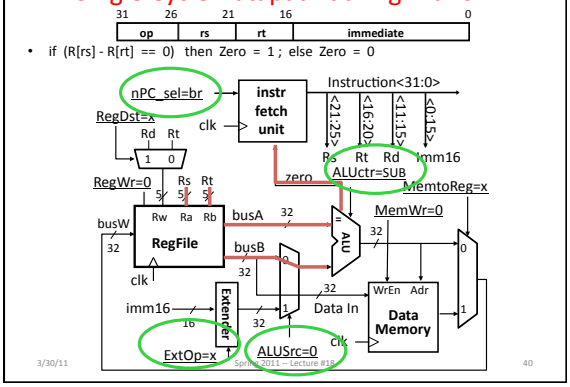
Single Cycle Datapath during Load



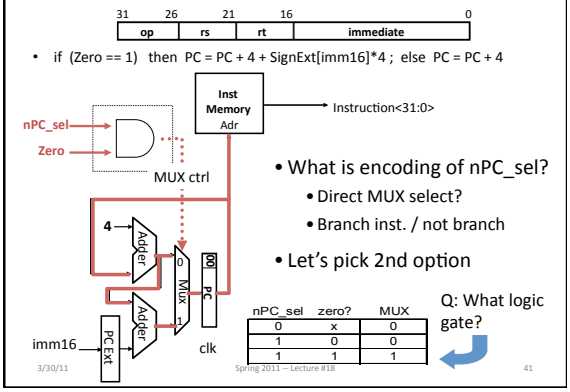
Single Cycle Datapath during Branch



Single Cycle Datapath during Branch

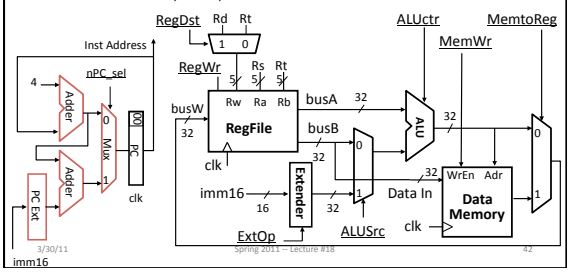


Instruction Fetch Unit at the End of Branch



Summary: Datapath's Control Signals

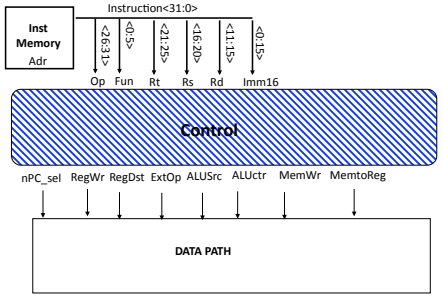
- ExtOp: "zero", "sign"
- ALUsrc: 0 ⇒ regB; 1 ⇒ immed
- ALUctr: "ADD", "SUB", "OR"
- MemWr: 1 ⇒ write memory
- MemtoReg: 0 ⇒ ALU; 1 ⇒ Mem
- RegDst: 0 ⇒ "rt"; 1 ⇒ "rd"
- RegWr: 1 ⇒ write register



Agenda

- MIPS-lite Datapath
- Administrivia
- CPU Timing
- MIPS-lite Control
- Datapath Control
- Technology Break
- Control Implementation

Given Datapath: RTL → Control



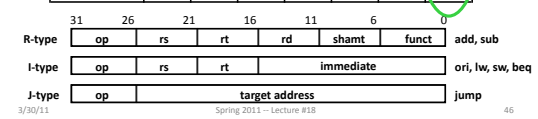
Summary of the Control Signals (1/2)

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"
lw	R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4
	ALUsrc=Im, Extop="sn", ALUctr="ADD", MemtoReg, RegDst=rt, RegWr, nPC_sel = "+4"
sw	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
	nPC_sel = "br", ALUctr = "SUB"

Summary of the Control Signals (2/2)

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	?
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	x



Boolean Expressions for Controller

```
RegDst    = add + sub
ALUSrc    = ori + lw + sw
MemtoReg  = lw
RegWrite  = add + sub + ori + lw
MemWrite  = sw
nPCsel    = beq
Jump      = jump
ExtOp     = lw + sw
ALUctr[0] = sub + beq   (assume ALUctr is  00 ADD,  01: SUB,  10: OR)
ALUctr[1] = or
```

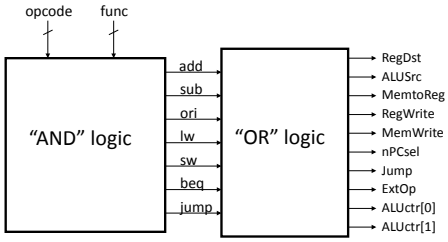
Where:

$$\begin{aligned}
\text{rtype} &= \sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet \sim op_1 \bullet \sim op_0,\\
\text{ori} &= \sim op_5 \bullet \sim op_4 \bullet op_3 \bullet op_2 \bullet \sim op_1 \bullet op_0\\
\text{lw} &= op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0\\
\text{sw} &= op_5 \bullet \sim op_4 \bullet op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0\\
\text{beq} &= \sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet op_2 \bullet \sim op_1 \bullet \sim op_0\\
\text{jump} &= \sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet \sim op_0
\end{aligned}$$

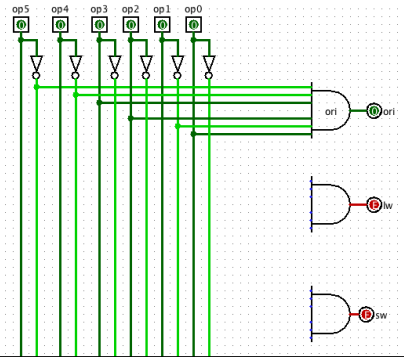
How do we implement this in gates?

$$\begin{aligned}
\text{add} &= \text{rtype} \bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet \sim func_1 \bullet \sim func_0\\
\text{sub} &= \text{rtype} \bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet func_1 \bullet \sim func_0
\end{aligned}$$

Controller Implementation

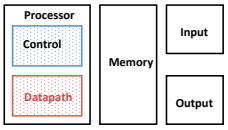


AND Control in Logisim

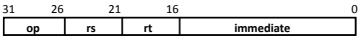


Summary: Single-cycle Processor

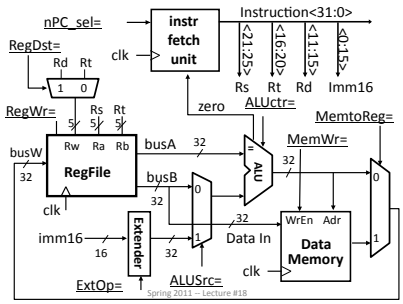
- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



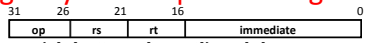
Single Cycle Datapath during Store



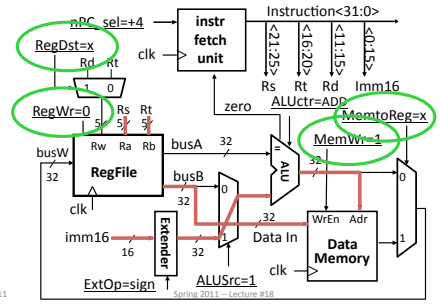
- Data Memory {R[rs] + SignExt[imm16]} = R[rt]



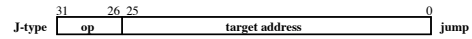
Single Cycle Datapath during Store



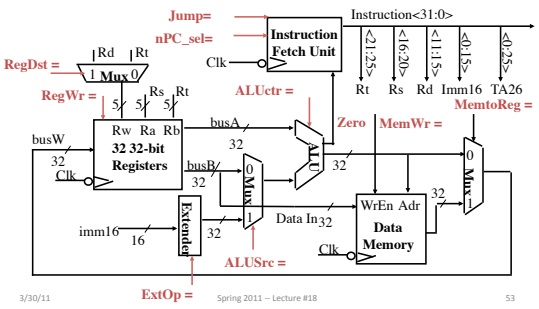
- Data Memory {R[rs] + SignExt[imm16]} = R[rt]



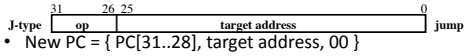
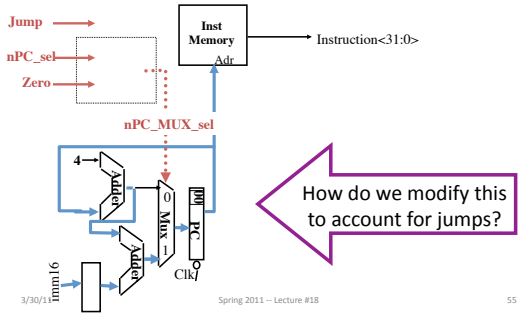
Single Cycle Datapath during Jump



- New PC = { PC[31..28], target address, 00 }



Instruction Fetch Unit at the End of Jump



- New PC = { PC[31..28], target address, 00 }