Csci 136 Computer Architecture II
–Single-Cycle Datapath

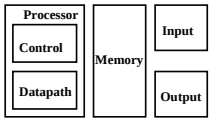
Xiuzhen Cheng
cheng@gwu.edu

Announcements

- **Homework Assignment #7 is due on March 10, before class.**
  - Readings: Sections 5.1-5.4
  - Problems: 5.1, 5.2, 5.8-5.10, 5.13, 5.28.
- **Project #2 is due on 11:59PM, March 10.**
- **Quiz #3: March 29, 2005**

The Big Picture

- **The Five Classic Components of a Computer**



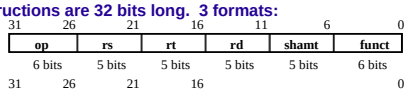
- **Performance of a machine is determined by:**
  - **Instruction count; Clock cycle time; Clock cycles per instruction**
- **Processor design (datapath and control) will determine:**
  - Clock cycle time; Clock cycles per instruction
  - Who will determine Instruction Count?
    - Compiler, ISA

How to Design a Processor: Step by Step

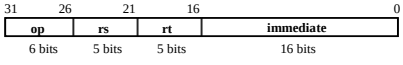
1. **Analyze instruction set => datapath requirements**
   1. the meaning of each instruction is given by the *register transfers*
   2. datapath must include storage element for registers
   3. datapath must support each register transfer
2. **Select the set of datapath components and establish clocking methodology**
3. **Assemble the datapath meeting the requirements**
4. **Analyze the implementation of each instruction to determine the settings of the control points that effects the register transfer**
5. **Assemble the control logic**

MIPS Instruction Format

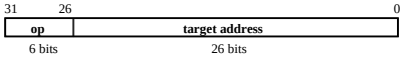
- **All MIPS instructions are 32 bits long. 3 formats:**
  - R-type



  - I-type



  - J-type



- **The different fields are:**
  - op: operation ("opcode") of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the "op" field
  - address / immediate: address offset or immediate value
  - target address: target address of jump instruction

MIPS Instruction Subset for Today

- **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

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)]<<2)
	else PC <– PC + 4

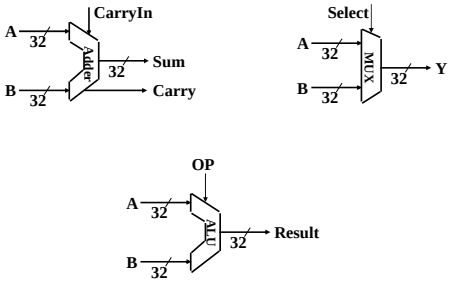
Step 1: Requirements of the Instruction Set

- **Memory**
  - instruction & data: instruction=MEM[PC]
- **Registers (32 x 32)**
  - read RS; read RT; Write RT or RD
- **PC, what is the new PC?**
- **Extender: sign-extension or 0-extension?**
- **Add and Sub register or extended immediate**
- **Add 4 or extended immediate to PC**

Step 2: Components of the Datapath

- **Combinational Elements**
- **Storage Elements**
  - Clocking methodology

Combinational Logic Elements (Basic Building Blocks)

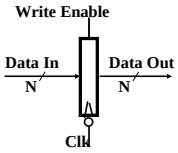


Basic Hardware Review

- **D Latch**
- **D Flip Flop**

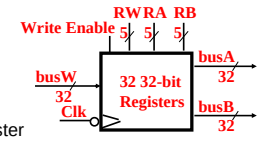
Storage Element: Register (Basic Building Block)

- **Similar to the D Flip Flop except**
  - N-bit input and output
  - Write Enable input
- **Write Enable:**
  - negated (0): Data Out will not change
  - asserted (1): Data Out will become Data In



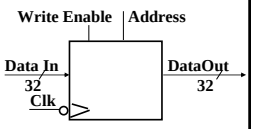
Storage Element: Register File

- **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 high
- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as combinational logic:
    - RA or RB valid => busA or busB outputs valid after "access time."



Storage Element: Idealized Memory

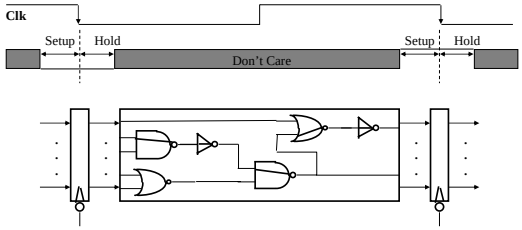
- **Memory (idealized)**
  - One input bus: Data In
  - One output bus: Data Out



- **Memory word is selected 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)**
  - The 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."

Clocking Methodology



- **All storage elements are clocked by the same clock edge**
- **Clock Cycle Time must be >=**

CLK-to-Q + Longest Delay Path + Setup Time + Clock Skew

- **Hold Time is also effected by clock skew**

Need enough time for signal to propagate through

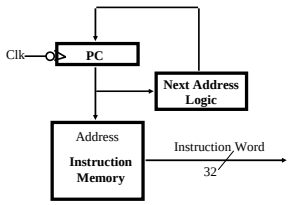
Step 3: Assemble DataPath meeting our requirements

- **Instruction Fetch**
  - Instruction = MEM[PC]
  - Update PC

- **Read Operands and Execute Operation**
  - Read one or two registers
  - Execute operation

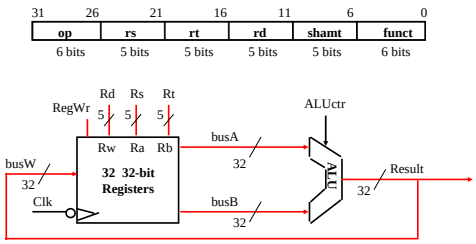
Datapath for Instruction Fetch

- **Fetch the Instruction: mem[PC]**
- **Update the program counter:**
  - Sequential Code: PC <- PC + 4
  - Branch and Jump: PC <- "something else"



Datapath for R-Type Instructions

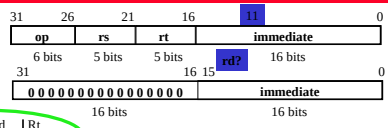
- **R[rd] <- R[rs] op R[rt] Example: addU rd, rs, rt**
  - Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
  - ALUctr and RegWr: control logic after decoding the instruction

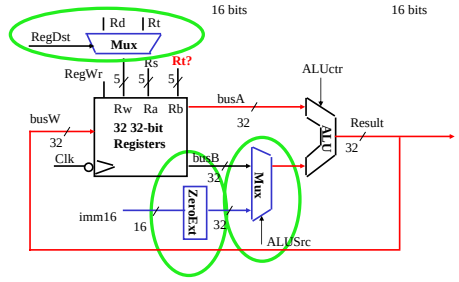


Logic Operations with Immediate

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

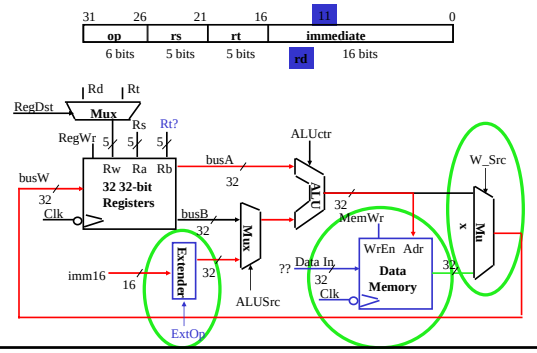
Eg. Ori $7, $8, 0x20





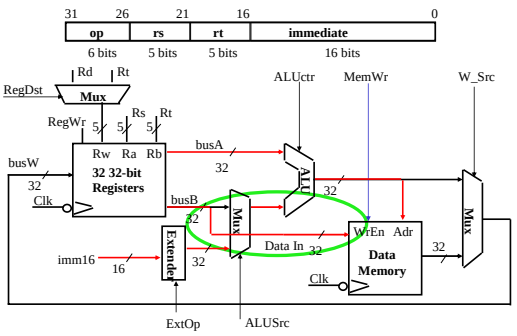
Load Operations

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

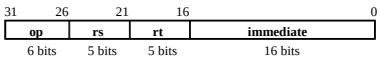


Store Operations

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



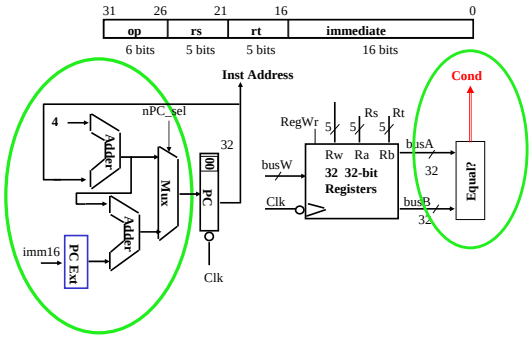
The Branch Instruction



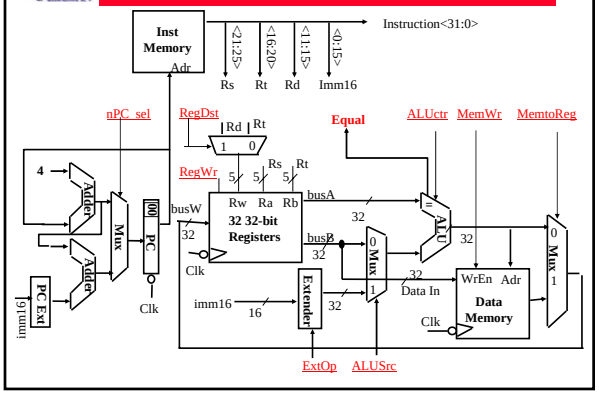
- beq rs, rt, imm16
  - mem[PC] Fetch the instruction from memory
  - Equal <- R[rs] == R[rt] Calculate the branch condition
  - if (Equal) Calculate the next instruction's address
    - PC <- PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC <- PC + 4

Datapath for Branch Operations

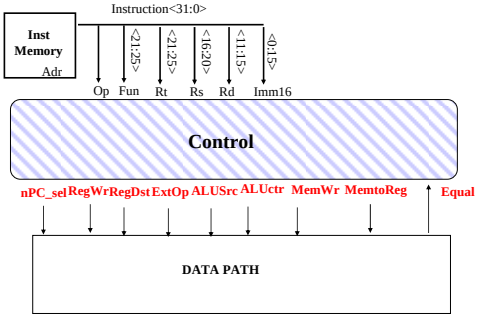
- beq rs, rt, imm16 Datapath generates condition (equal)



Putting it All Together: A Single Cycle Datapath

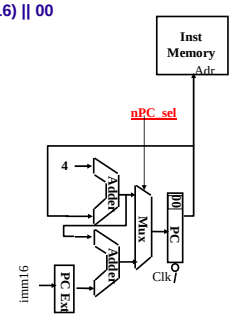


Step 4: Given Datapath: RTL -> Control



Meaning of the Control Signals

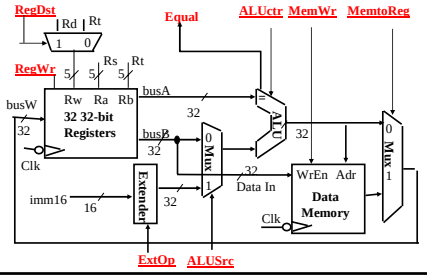
- Rs, Rt, Rd and Imed16 hardwired into datapath
- nPC_sel: 0 => PC <– PC + 4; 1 => PC <– PC + 4 + SignExt(Im16) || 00



Meaning of the Control Signals

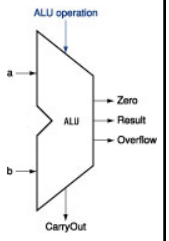
- ExtOp: "zero", "sign"
- ALUsrc: 0 => regB; 1 => immed
- ALUctr: "add", "sub", "or"

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



Review on ALU Design

ALU Control Lines	Function
0000	And
0001	Or
0010	Add
0110	Subtraction
0111	Slt, beq
1100	NOR



ALU Control and the Central Control

- **Two-level design to ease the job**
  - ALU Control generates the 4 control lines for ALU operation
  - Func code field is only effective for R-type instructions, whose Opcode field contains 0s.
  - The operation of I-type and J-type instructions is determined only by the 6 bit Opcode field.
  - Lw/sw and beq need ALU even though they are I-type instructions.
  - Three cases: address computation for lw/sw, comparison for beq, and R-Type; needs two control lines from the main control unit: ALUOp: 00 for lw/sw, 01 for beq, 10 for R-type
- **Design ALU control**
  - Input: the 6 bit func code field for R-type
  - Input: the 2 bit ALUOp from the main control unit.
- **Design the main control unit**
  - Input: the 6 bit Opcode field.

Fig 5.15 (LW/SW, ALU Operatios, BEQ)

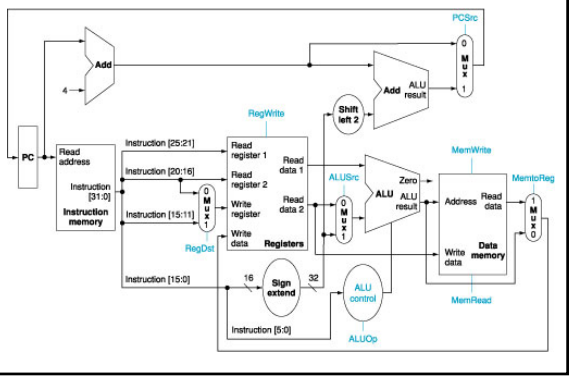
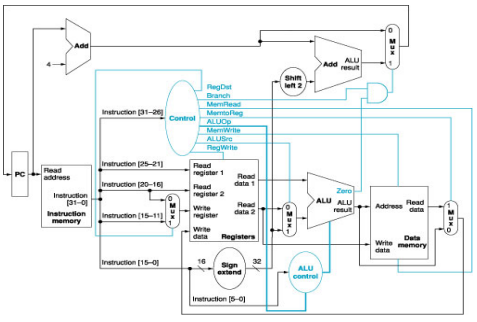


Fig 5.17 (LW/SW, ALU Operatios, BEQ)



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 = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__	
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel =__	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel = __	

Control Signals (Answer)

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	
	nPC_sel = EQUAL, ALUctr = “sub”	

Step 5: Logic for each control signal

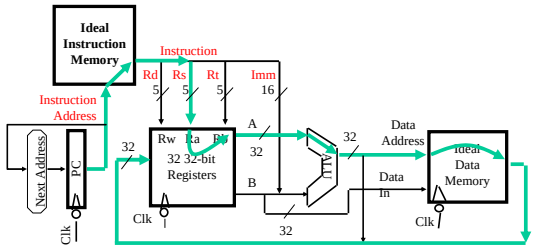
- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == “000000”) then “regB” else “immed”
- ALUctr <= if (OP == “000000”) then funct
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- ExtOp <= _____
- MemWr <= _____
- MemtoReg <= _____
- RegWr: <= _____
- RegDst: <= _____

Step 5: Logic for each control signal (Answer)

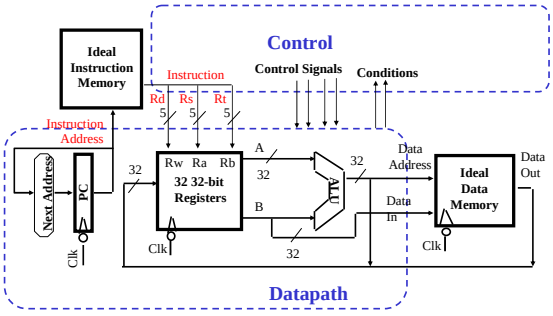
- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == “000000”) then “regB” else “immed”
- ALUctr <= if (OP == “000000”) then funct
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- ExtOp <= if (OP == ORi) then “zero” else “sign”
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

An Abstract View of the Critical Path

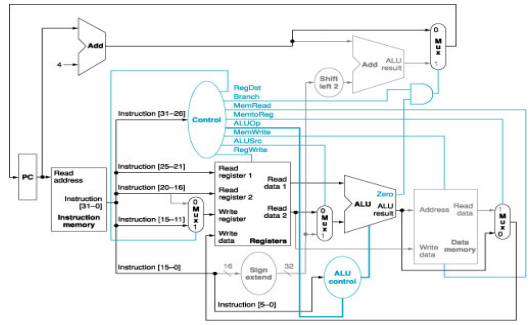
- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:



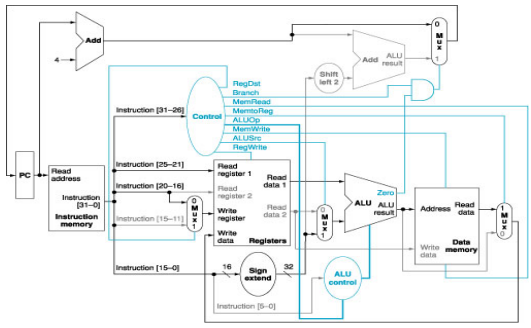
An Abstract View of the Implementation



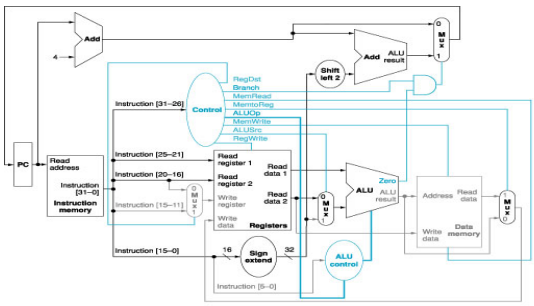
Example: R-type add $t1, $t2, $t3



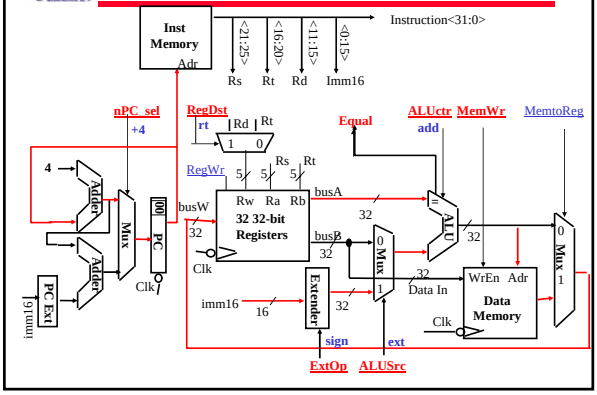
Example: lw



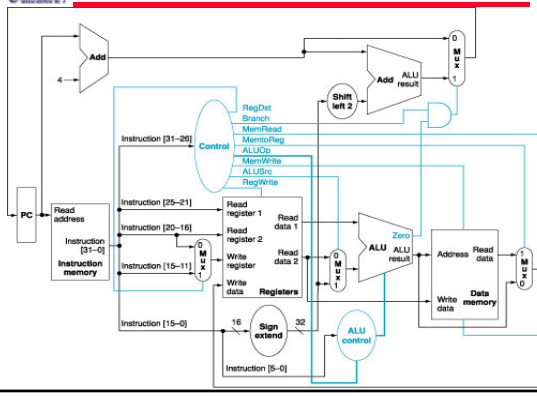
Example: beq



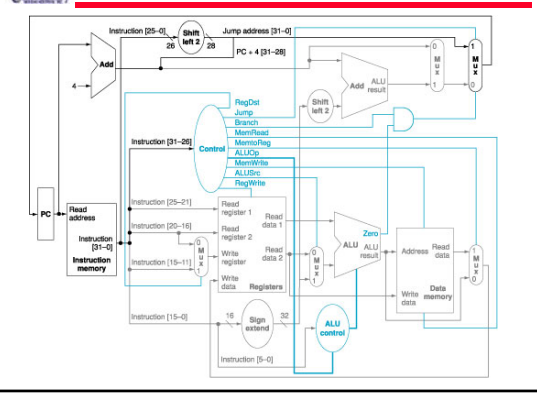
Example: Load Instruction



How to Implement jump Instruction?



How to Implement J Answer



Performance of Single-Cycle Datapath

- **Time needs by functional units:**
  - Memory units: 200 ps
  - ALU and adders: 100 ps
  - Register file (r/w): 50 ps
  - No delay for other units
- **Two single cycle datapath implementations**
  - Clock cycle time is the same for all instructions
  - Variable clock cycle time per instruction
- **Instruction mix: 25% loads, 10% stores, 45% ALU, 15% branches, and 5% jumps**
- **Compare the performance of R-type, lw, sw, branch, and j**
- **Answer can be found at page 315-316**

Single-Cycle Processor

- **Advantage**
  - One clock cycle per instruction
- **Disadvantage**
  - Clock cycle is long

Summary

- **5 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 affects the register transfer**
  - **5. Assemble the control logic**
- **MIPS makes it easier**
  - **Instructions same size**
  - **Source registers always in same place**
  - **Immediates same size, location**
  - **Operations always on registers/immediates**
- **Single cycle datapath => CPI=1, CCT => long**