CS4100: 計算機結構

Designing a Single-Cycle Processor

國立清華大學資訊工程學系
九十三學年度第一學期

Adapted from class notes of D. Patterson

Outline

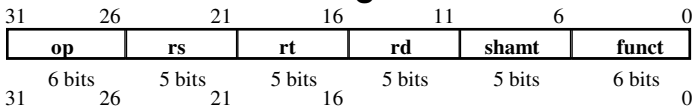
- Designing a processor
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

How to Design a Processor?

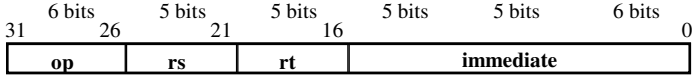
1. Analyze instruction set (datapath requirements)
   - The meaning of each instruction is given by the *register transfers*
   - Datapath must include storage element
   - Datapath must support each register transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points effecting register transfer
5. Assemble the control logic

Step 1: Analyze Instruction Set

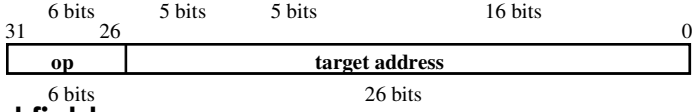
- All MIPS instructions are 32 bits long with 3 formats:
  - R-type:



  - I-type:



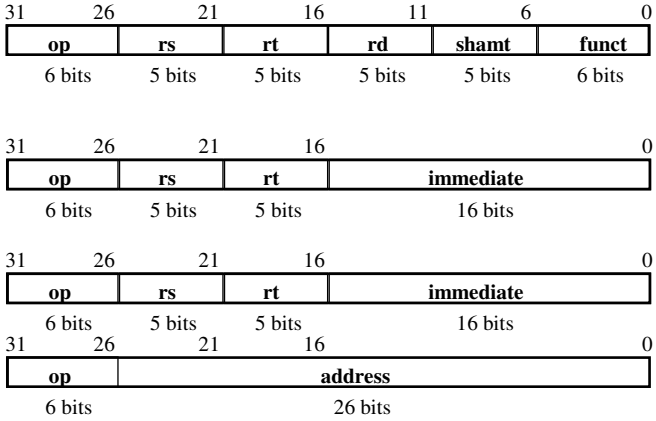
  - J-type:



- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: source and destination register
  - shamt: shift amount
  - funct: selects variant of the "op" field
  - address / immediate
  - target address: target address of jump

Our Example: A MIPS Subset

- R-Type:
  - add rd, rs, rt
  - sub rd, rs, rt
  - and rd, rs, rt
  - or rd, rs, rt
  - slt rd, rs, rt
- Load/Store:
  - lw rt,rs,imm16
  - sw rt,rs,imm16
- Branch:
  - beq rs,rt,imm16
- Jump:
  - j target



Logical Register Transfers

- RTL gives the meaning of the instructions
- All start by fetching the instruction, read registers, then use ALU => simplicity and regularity help

MEM[PC] = op | rs | rt | rd | shamt | funct
or = op | rs | rt | Imm16
or = op | Imm26

Inst	Register transfers
ADD	R[rd] <- R[rs] + R[rt]; PC <- PC + 4
SUB	R[rd] <- R[rs] - R[rt]; 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

Requirements of Instruction Set

After checking the register transfers, we can see that datapath needs the followings:

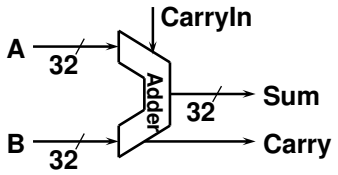
- Memory
  - store instructions and data
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Extender for zero- or sign-extension
- Add and sub register or extended immediate
- Add 4 or extended immediate to PC

Outline

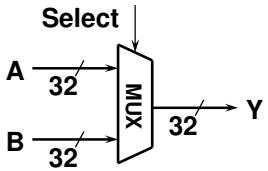
- Designing a processor
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

Step 2a: Datapath Components

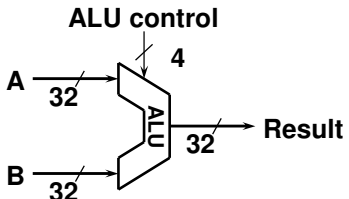
- Basic building blocks of combinational logic elements :



Adder



MUX

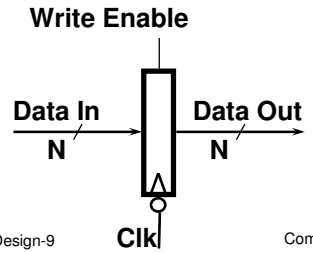


ALU

Step 2b: Datapath Components

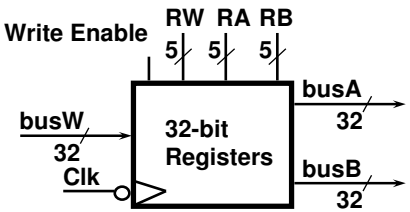
Storage elements:

- Register:
  - 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

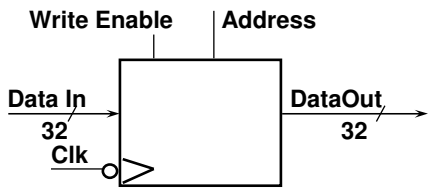
- Consists of 32 registers:
  - Appendix B.8
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW



- Register is selected by:
  - RA selects the register to put on busA (data)
  - RB selects the register to put on busB (data)
  - RW selects the register to be written via busW (data) when Write Enable is 1
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational circuit

Storage Element: Memory

- Memory (idealized)
  - Appendix B.8
  - One input bus: Data In
  - One output bus: Data Out



- 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
    - No need for read control

Step 3a: Datapath Assembly

- Instruction fetch unit: common operations
  - Fetch the instruction: mem[PC]
  - Update the program counter:
    - Sequential code: PC <- PC + 4
    - Branch and Jump: PC <- "Something else"

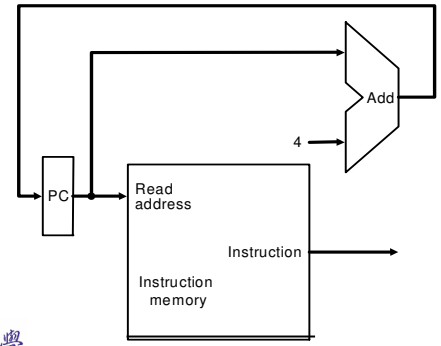


Fig. 5.6

Step 3b: Add and Subtract

- R[rd] <- R[rs] op R[rt] Ex: add rd, rs, rt
  - Ra, Rb, Rw come from inst.'s rs, rt, and rd fields
  - ALU and RegWrite: control logic after decode

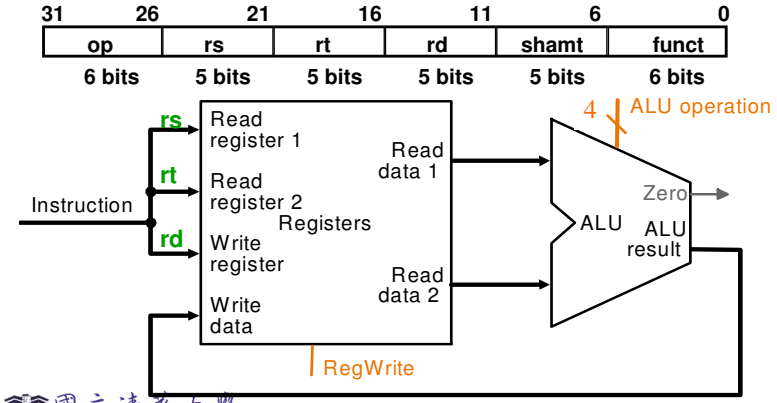


Fig. 5.7

Step 3c: Load/Store Operations

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

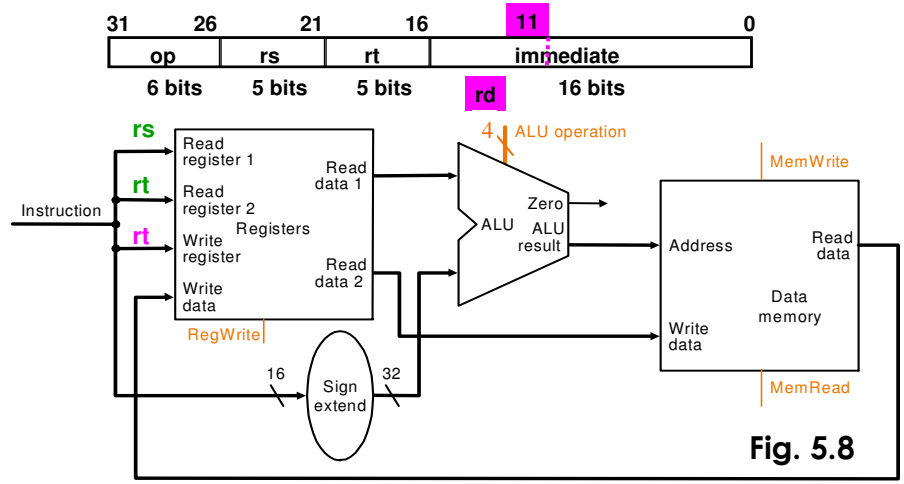


Fig. 5.8

Datapath for Memory and R-type (b+c)

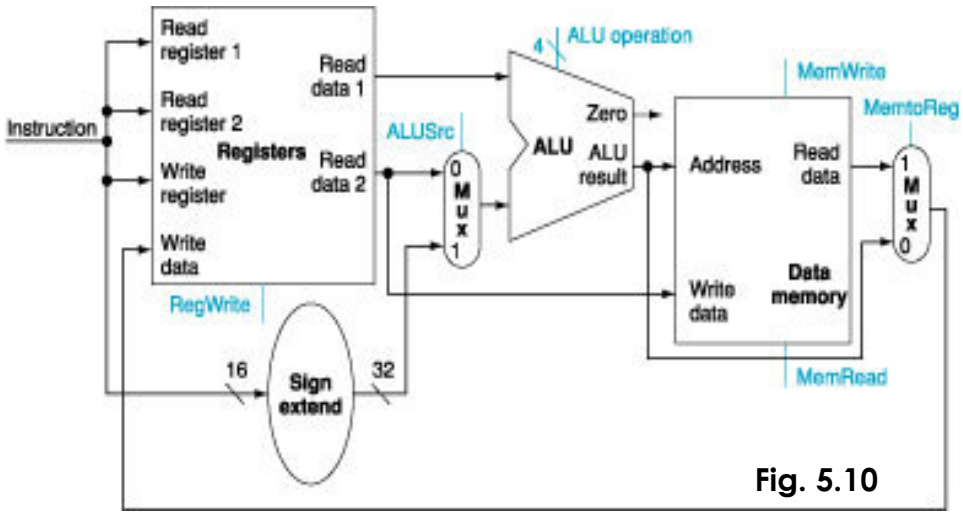


Fig. 5.10

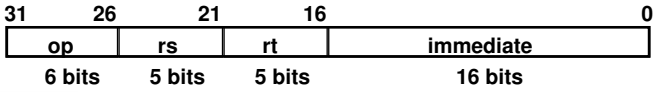
Step 3d: Branch Operations

- beq rs, rt, imm16

```
mem[PC]                 Fetch inst. from memory

Equal <- R[rs] == R[rt] Calculate branch condition

if (COND == 0)          Calculate next inst. address
    PC <- PC + 4 + ( SignExt(imm16) x 4 )
else
    PC <- PC + 4
```



Datapath for Branch Operations

- beq rs, rt, imm16

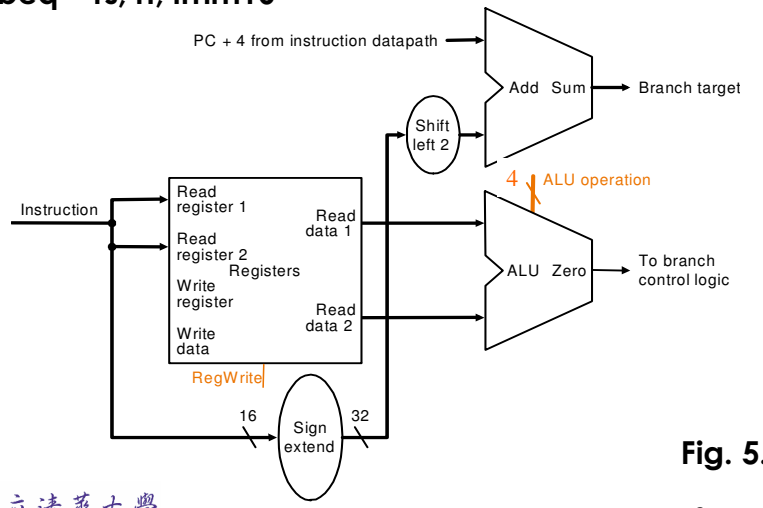


Fig. 5.9

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

A Single Cycle Datapath

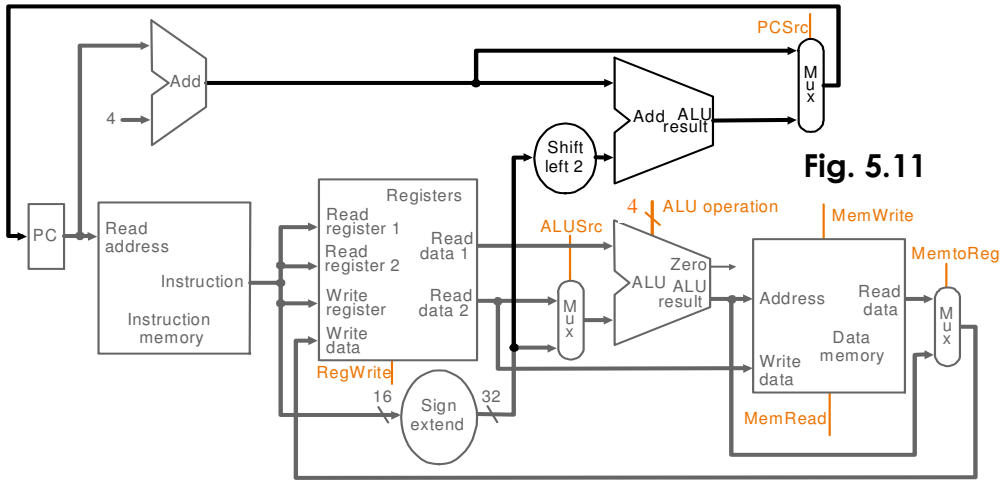
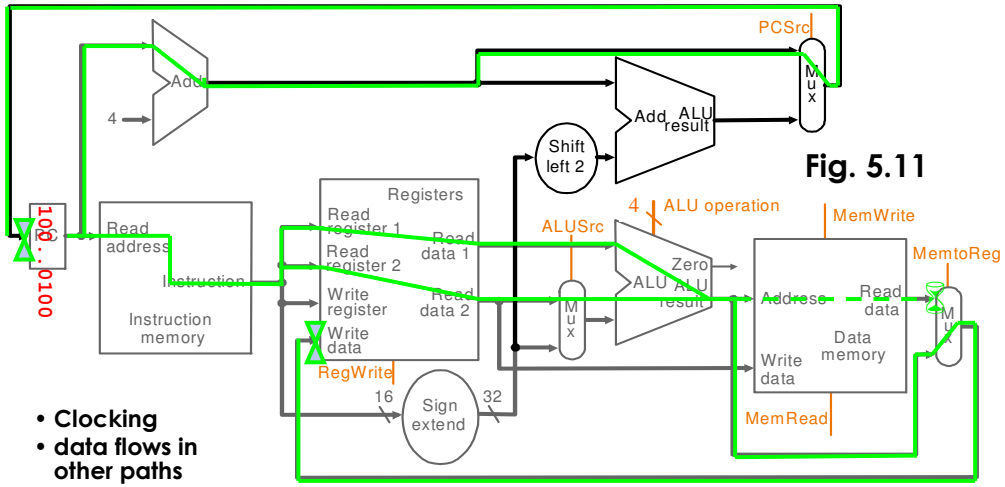


Fig. 5.11

Data Flow during add

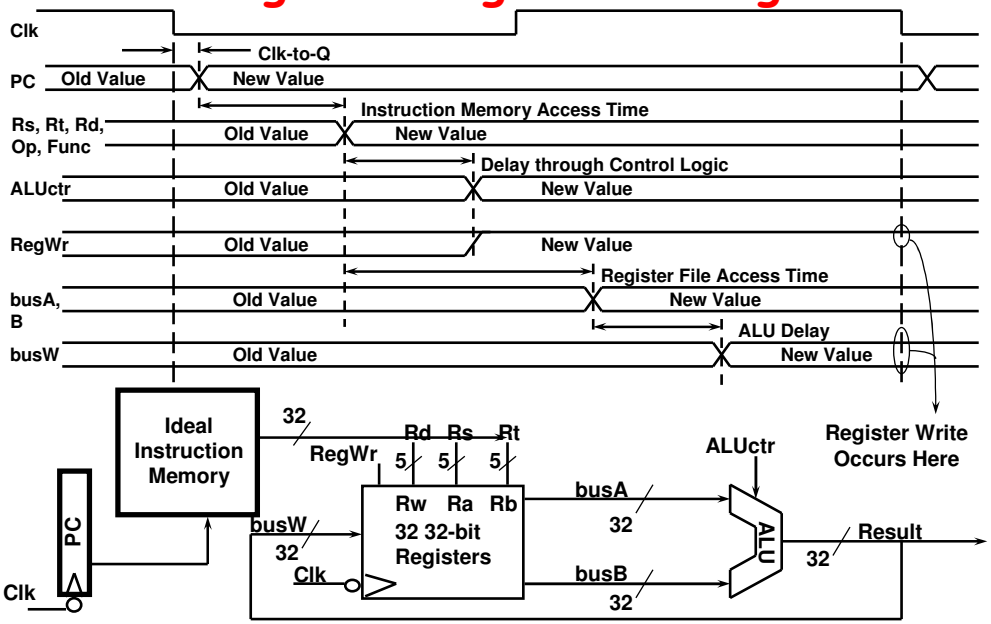


- Clocking
- data flows in other paths

Clocking Methodology

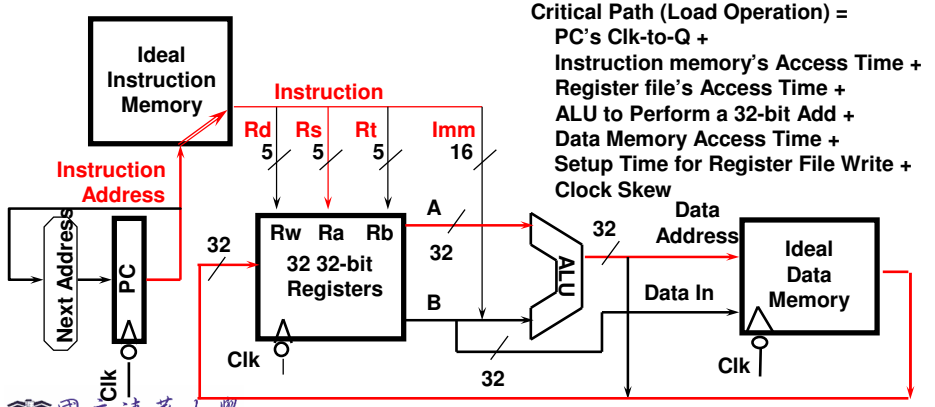
- Define when signals are read and written
- Assume edge-triggered:
  - Values in storage (state) elements updated only on a clock edge
    => clock edge should arrive only after input signals stable
  - Any combinational circuit must have inputs from and outputs to storage elements
  - *Clock cycle*: time for signals to propagate from one storage element, through combinational circuit, to reach the second storage element
  - A register can be read, its value propagated through some combinational circuit, new value is written back to the same register, all in same cycle => no feedback within a single cycle

Register-Register Timing



The Critical Path

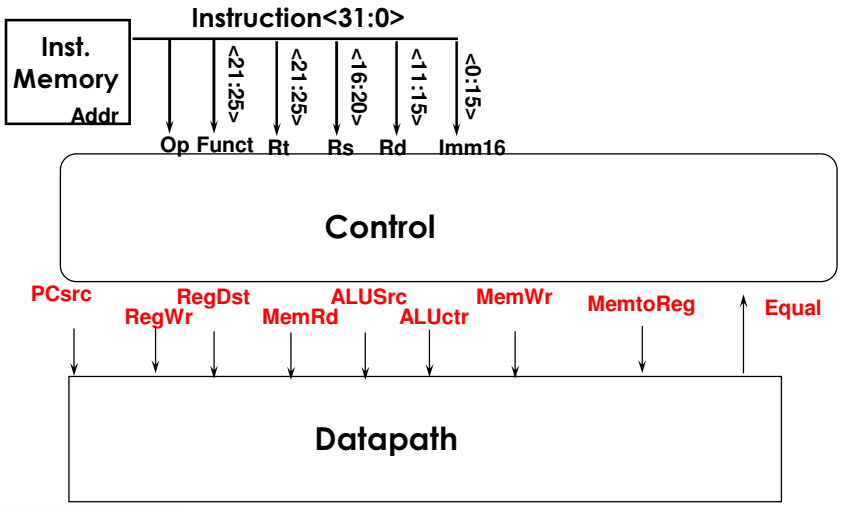
- Register file and ideal memory:
  - During read, behave as combinational logic:
    - Address valid => Output valid after access time



Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

Step 4: Control Points and Signals



Designing Main Control

- Some observations:
  - opcode (Op[5-0]) is always in bits 31-26
  - two registers to be read are always in rs (bits 25-21) and rt (bits 20-16) (for R-type, beq, sw)
  - base register for lw and sw is always in rs (25-21)
  - 16-bit offset for beq, lw, sw is always in 15-0
  - destination register is in one of two positions:
    - lw: in bits 20-16 (rt)
    - R-type: in bits 15-11 (rd)

    => need a multiplex to select the address for written register

Datapath with Mux and Control

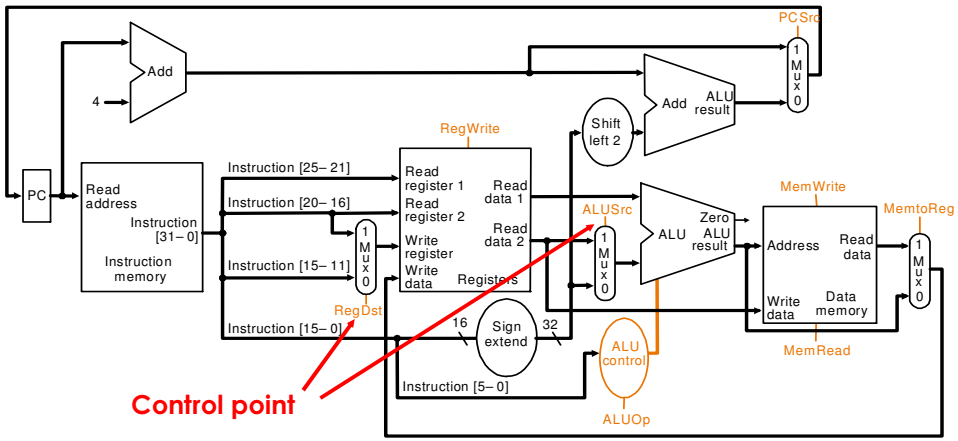
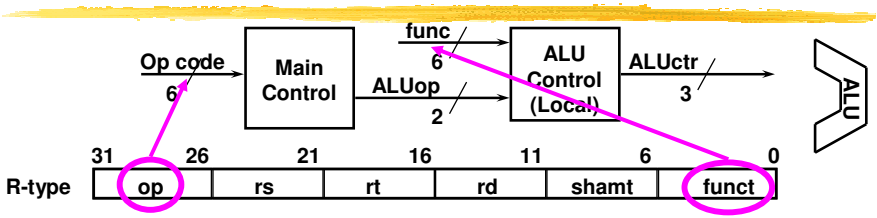


Fig. 5.15

Our Plan for the Controller



- ALUop is 2-bit wide to represent:
  - "I-type" requiring the ALU to perform:
    - (00) add for load/store and (01) sub for beq
  - "R-type" (10, need to reference `func` field

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

Datapath with Control Unit

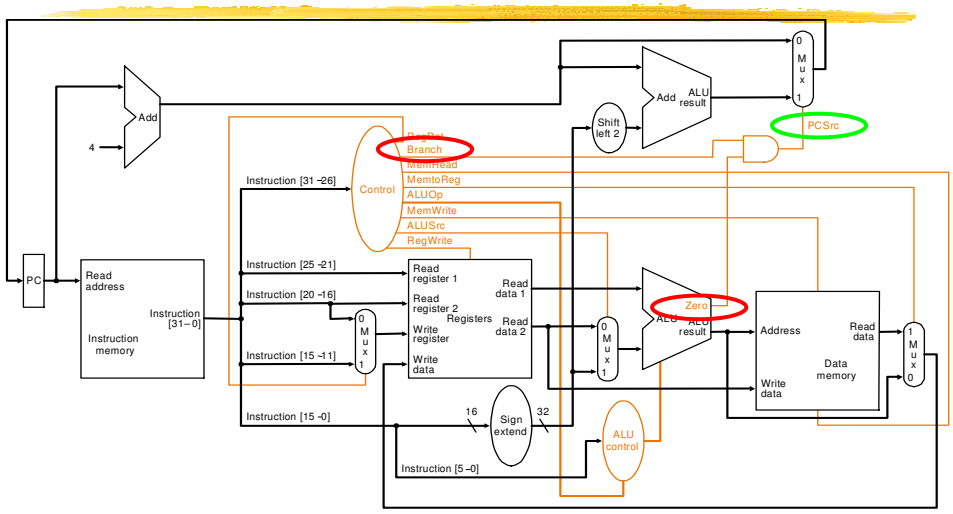
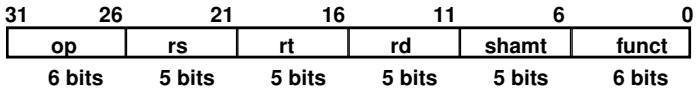


Fig. 5.17

Operation of Datapath: add



- add rd, rs, rt

mem[PC] PC+4	1. Fetch the instruction from memory
R[rs], R[rt]	2. Instruction decode and read operands
R[rs] + R[rt]	3. Execute the actual operation
R[rd] <- ALU PC <- PC+4	4. Write back to target register

Instruction Fetch at Start of `Add`

- instruction <- mem[PC]; PC + 4

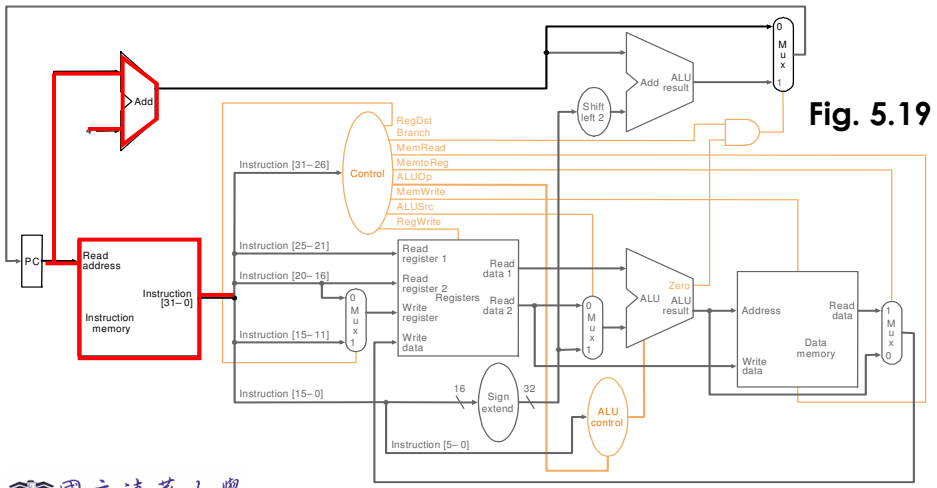


Fig. 5.19

Instruction Decode of Add

- Fetch the two operands and decode instruction:

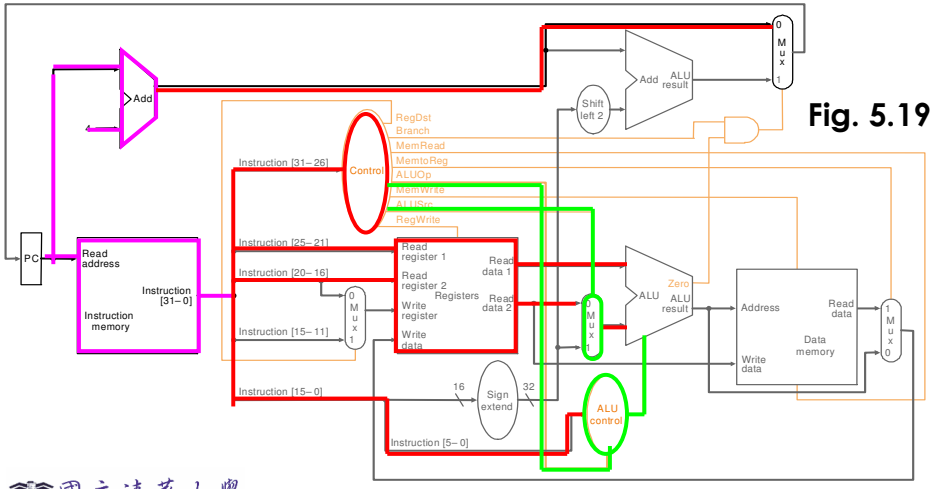


Fig. 5.19

ALU Operation during Add

- R[rs] + R[rt]

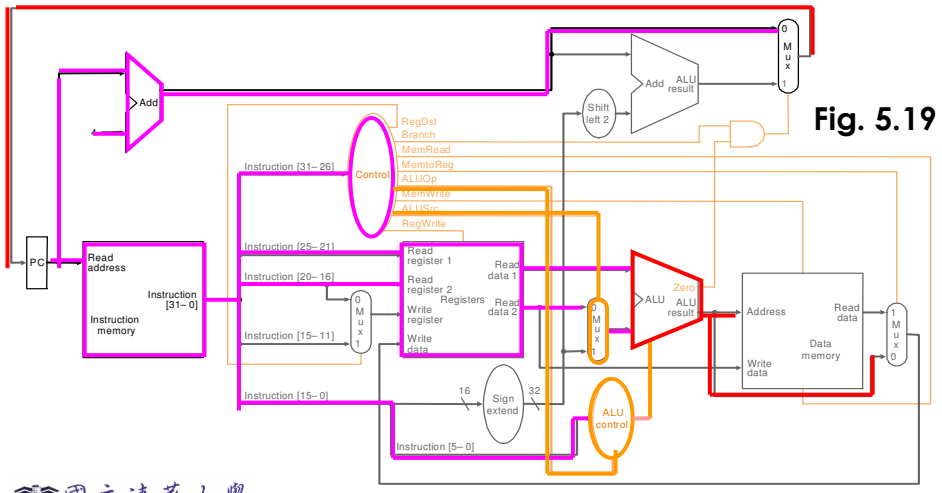


Fig. 5.19

Write Back at the End of Add

- R[rd] <- ALU; PC <- PC + 4

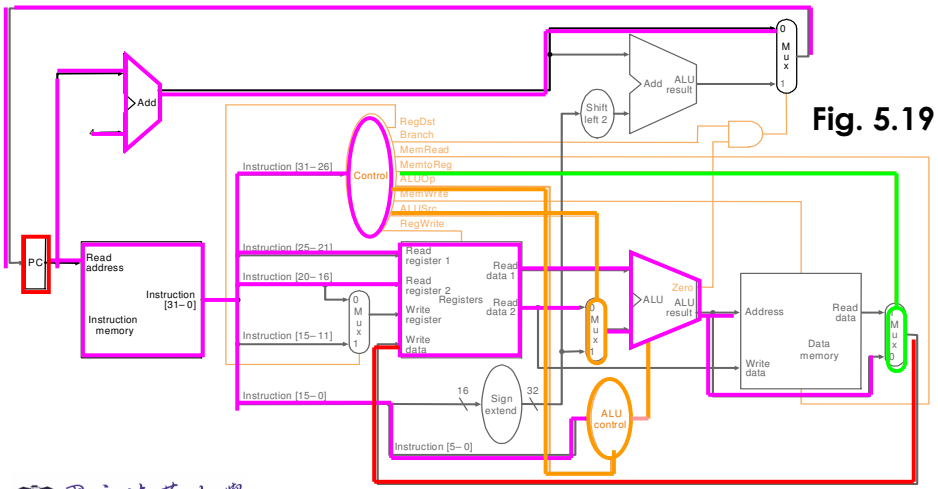


Fig. 5.19

Datapath Operation for lw

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

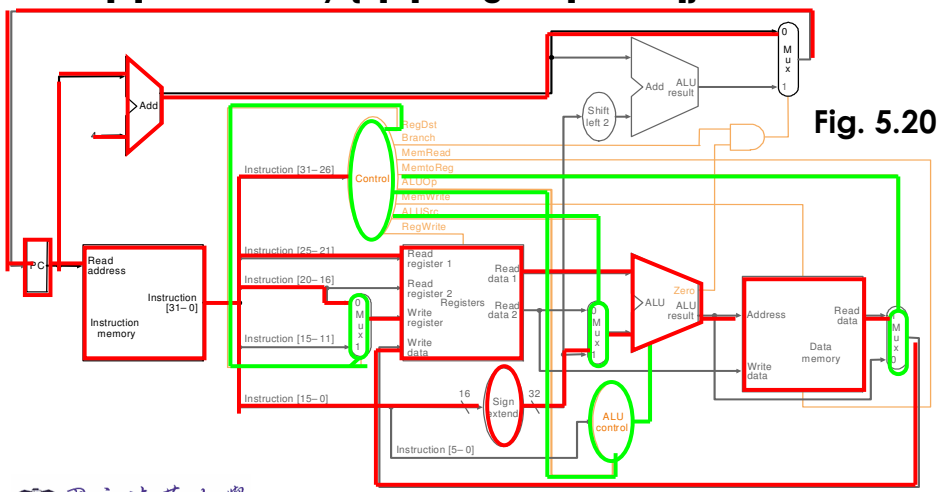


Fig. 5.20

Datapath Operation for beq

if (R[rs]-R[rt]==0) then Zero<-1 else Zero<-0
if (Zero==1) then PC=PC+4+signExt[imm16]*4; else PC = PC + 4

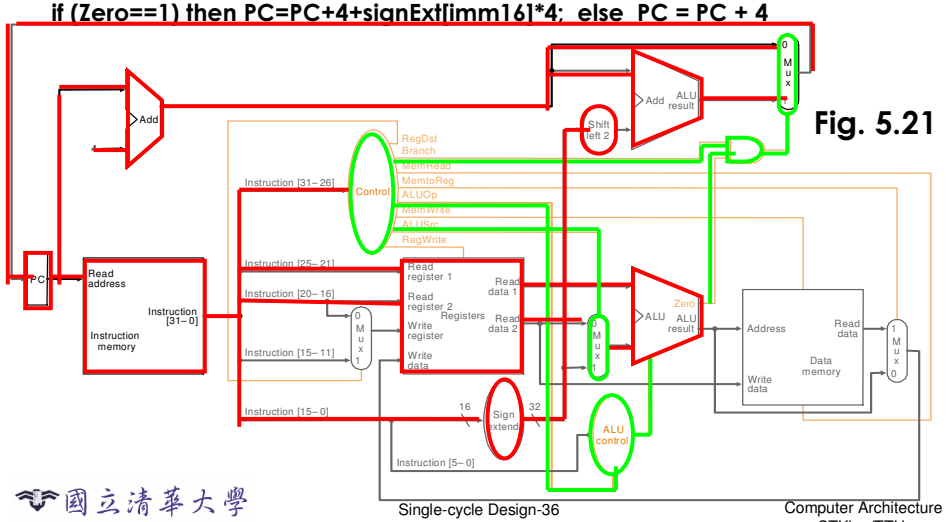


Fig. 5.21

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

Control Signals

RegDst: 0=>rt; 1=>rd RegWrite: 1=>write dest. reg.
ALUsrc: 0=>regB; PCsrc: 0=>PC<-PC+4;
1=>immed 1=>PC<-branch addr.
MemRd:1=>read memory MemWr:1=>write memory
MemtoReg: 0=>write reg. from ALU;
1=>write reg. from memory (Fig. 5.16)

inst Register Transfer
ADD R[rd] <- R[rs] + R[rt]; PC <- PC + 4
ALUsrc = RegB, ALUctr = "add", RegDst = rd, RegWr, PCsrc = "+4"
SUB R[rd] <- R[rs] - R[rt]; PC <- PC + 4
ALUsrc = RegB, ALUctr = "sub", RegDst = rd, RegWr, PCsrc = "+4"
LOAD R[rt] <- MEM[R[rs] + sign_ext(Imm16)]; PC <- PC + 4
ALUsrc = Im, ALUctr = "add", MemtoReg, RegDst = rt, RegWr, PCsrc="+4"
STORE MEM[R[rs] + sign_ext(Imm16)] <- R[rs]; PC <- PC + 4
ALUsrc = Im, ALUctr = "add", MemWr, PCsrc = "+4"
BEQ if (R[rs]==R[rt]) then PC<-PC+sign_ext(Imm16)] || 00 else PC<-PC+4
PCsrc = Branch address, ALUctr = "sub"

Step 5a: Implement ALU Control

- Recall: ALU design in Chapter 3 and func

ALUctr	ALU Operation
0000	AND
0001	OR
0010	add
0110	sub
0111	set-on-less-than

funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than

	ALUop		func						ALU	ALUctr			
	bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	Operation	bit<3>	bit<2>	bit<1>	bit<0>
lw	0	0	x	x	x	x	x	x	Add	0	0	1	0
sw	0	0	x	x	x	x	x	x	Add	0	0	1	0
beq	0	1	x	x	x	x	x	x	Subtract	0	1	1	0
R	1	0	1	0	0	0	0	0	Add	0	0	1	0
R	1	0	1	0	0	0	1	0	Subtract	0	1	1	0
R	1	0	1	0	0	1	0	0	And	0	0	0	0
R	1	0	1	0	0	1	0	1	Or	0	0	0	1
R	1	0	1	0	1	0	1	0	Set on <	0	1	1	1

Logic Equation for ALUctr

ALUop		func						ALUctr			
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>
0	0	x	x	x	x	x	x	0	0	1	0
x	1	x	x	x	x	x	x	0	1	1	0
1	x	x	x	0	0	0	0	0	0	1	0
1	x	x	x	0	0	1	0	0	1	1	0
1	x	x	x	0	1	0	0	0	0	0	0
1	x	x	x	0	1	0	1	0	0	0	1
1	x	x	x	1	0	1	0	0	1	1	1

Fig. 5.13

Logic Equation for ALUctr2

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
x	1	x	x	x	x	x	x	1
1	x	x	x	0	0	1	0	1
1	x	x	x	1	0	1	0	1

This makes func<3> a don't care

Fig. 5.13

ALUctr2 = ALUop0
+ ALUop1 • func2' • func1 • func0'

Logic Equation for ALUctr1

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	x	x	x	x	x	x	1
x	1	x	x	x	x	x	x	1
1	x	x	x	0	0	0	0	1
1	x	x	x	0	0	1	0	1
1	x	x	x	1	0	1	0	1

Fig. 5.13

ALUctr1 = ALUop1'
+ ALUop1 • func2' • func0'

Logic Equation for ALUctr0

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
1	x	x	x	0	1	0	1	1
1	x	x	x	1	0	1	0	1

Fig. 5.13

ALUctr0 = ALUop1
• func3' • func2 • func1' • func0
+ ALUop1' • func3
• func2' • func1 • func0'

- See Fig. 5.13 for complete truth table

The Resultant ALU Control Block

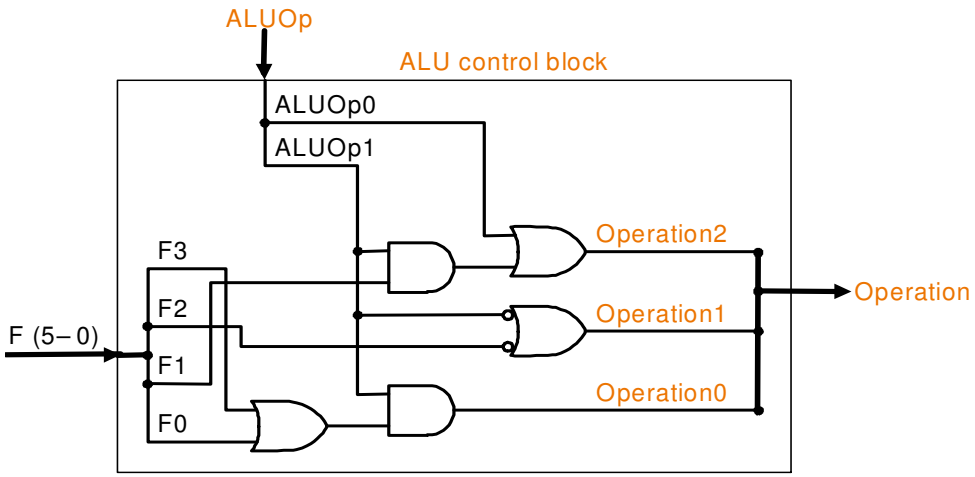


Fig. C.2.3

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

Step 5b: Implement Main Control

- Logic equation for each control signal:

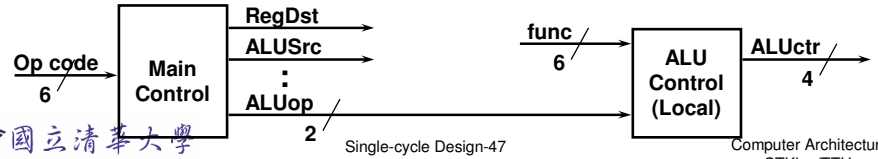
Branch: if (OP == BEQ) then 1 else 0
ALUsrc : if (OP == 0) then regB else immed
ALUop: if (OP == 0) then funct
elseif (OP == BEQ) then "sub"
else "add"
MemWr: (OP == SW)
MemtoReg: (OP == LW)
RegWr: if ((OP == SW) || (OP == BEQ))
then 0 else 1
RegDst: if (OP == LW) then 0 else 1

Truth Table of Control Signals

See Appendix A

	func: 10 0000	func: 10 0010	We Don't Care :-)		
	op: 00 0000	op: 00 0000	op: 10 0011	op: 10 1011	op: 00 0100
	add	sub	lw	sw	beq
RegDst	1	1	0	x	x
ALUSrc	0	0	1	1	0
MemtoReg	0	0	1	x	x
RegWrite	1	1	1	0	0
MemRead	0	0	1	0	0
MemWrite	0	0	0	1	0
Branch	0	0	0	0	1
ALUop1	1	1	0	0	0
ALUop0	0	0	0	0	1

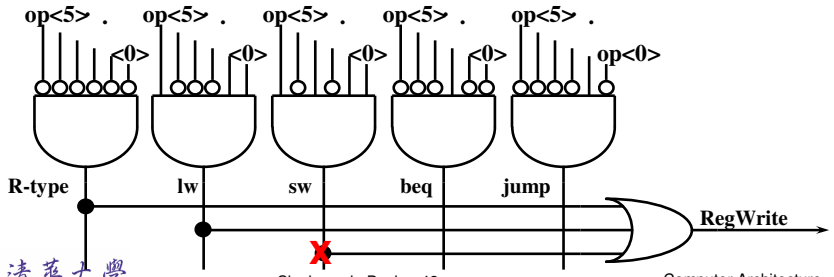
Fig. 5.22



Truth Table for RegWrite

Op code	00 0000	10 0011	10 1011	00 0100
	R-type	lw	sw	beq
RegWrite	1	1	0	0

RegWrite = R-type + lw
= op5' • op4' • op3' • op2' • op1' • op0' (R-type)
+ op5 • op4' • op3' • op2' • op1 • op0 (lw)



PLA Implementing Main Control

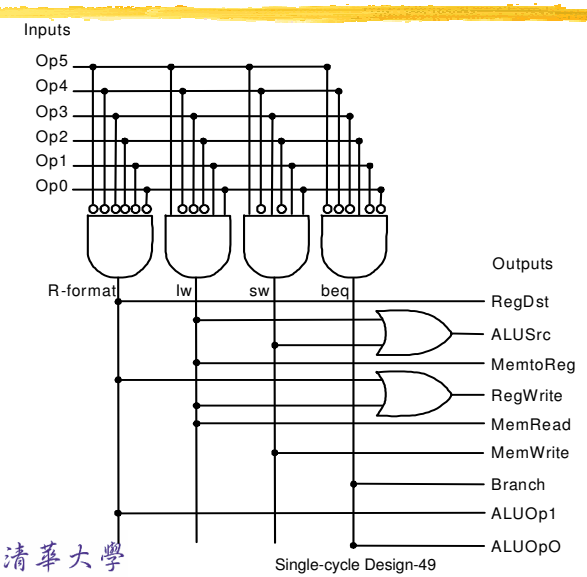


Fig. C.2.5

Putting it Altogether (+ jump instruction)

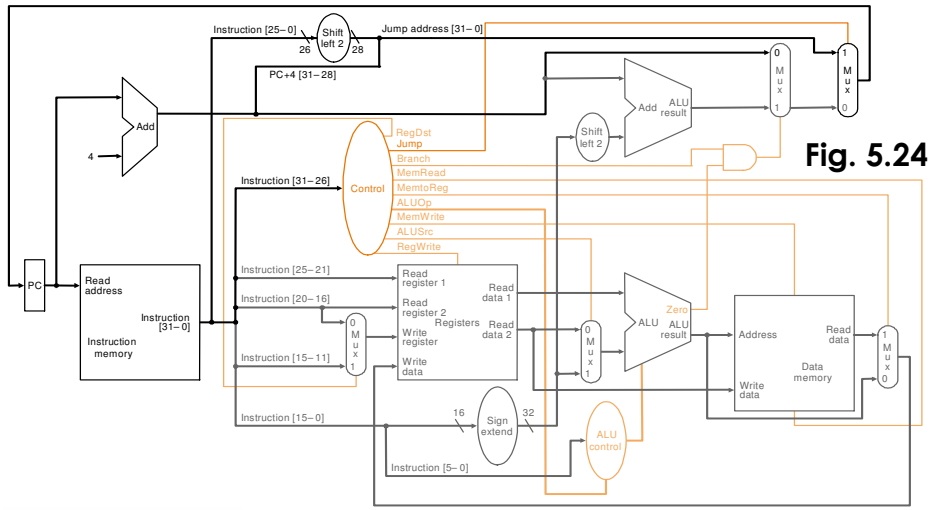
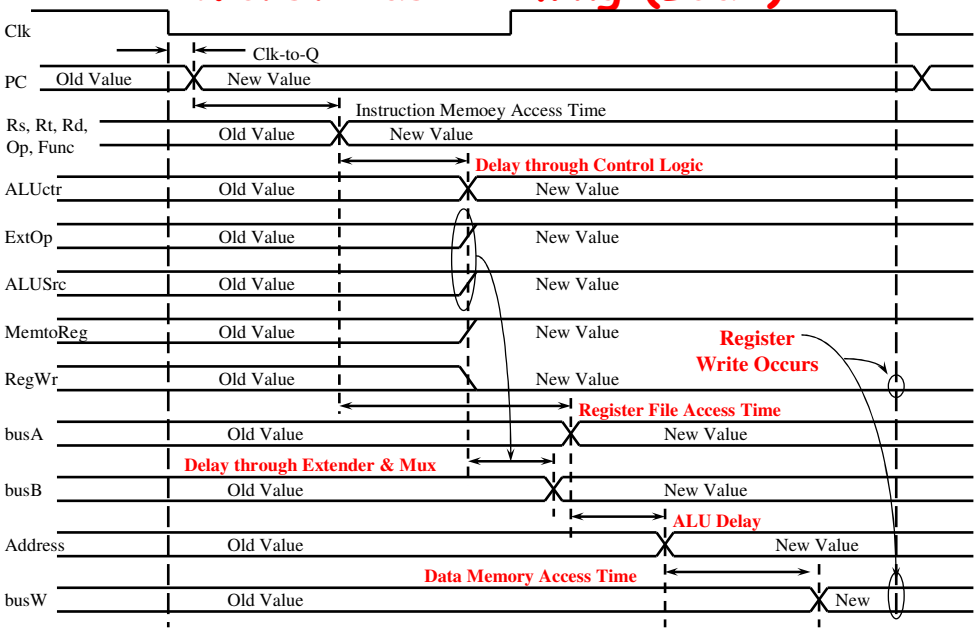


Fig. 5.24

Worst Case Timing (Load)



Drawback of Single-Cycle Design

- Long cycle time:
  - Cycle time must be long enough for the load instruction:
    PC's Clock -to-Q +
    Instruction Memory Access Time +
    Register File Access Time +
    ALU Delay (address calculation) +
    Data Memory Access Time +
    Register File Setup Time +
    Clock Skew
- Cycle time for load is much longer than needed for all other instructions

Summary

- Single cycle datapath => CPI=1, Clock cycle time long
- 5 steps to design a processor:
  1. Analyze ISA => datapath requirements
  2. Select set of datapath components
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points
  5. Assemble the control logic
- MIPS makes control easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates