MIPS, Data Path, and Assembly Language

JoJo – EZ4

MIPS(Microprocessor without Interlocked Pipeline Stages)

- Build in 1981 from Stanford U
- Used in Windows CE (before 2005), Nintendo 64(1996), Sony PlayStation, PlayStation 2 and PlayStation Portable (before 2003)
- Being studied in computer architecture courses at most Universities.

Instruction

- Computer need instruction to do computation (add, shift, etc)
- CPU finds the location and executes the instruction (**Fetch and Decode**)
- In 32-bit system, instruction is a **32-bit(4-Byte)** binary string, like

00000000 00000001 00111000 00100011 or (FFCA 078B)

Instruction Fetch

- **Program Counter (PC)** stores the location of the current instruction

- Each instruction is **4 byte** long, so we can do **+4 increment** to fetch instruction 1 by 1

- PC value can also be loaded from the result of ALU operation, so we can also jump to fetch a instruction not near the current location

MIPS Instruction(Decode)

- **32 bits** in total for every instruction

- Only **3 types** of instructions

Type	-31- format (bits) -0-					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

R-type Instruction

R-type

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

- Use three Registers as operands: two as **sources(Rs, Rt),** and one as a **destination(Rd).**

- Op-code(operation code) are 0 (6 个 0)

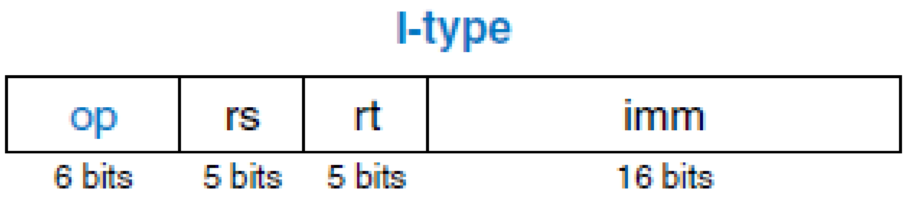
- Last 6 bit Funct indicating the **functionality** of the instruction.

- The fifth field, *shamt*, is used only in shift operations. In those instructions, the binary value stored in the 5-bit *shamt* field indicates the amount to shift. For all other R-type instructions, *shamt* is 0.

R-type instruction data path

- Instruction 从哪里来？
  - Instruction Memory
- Instruction 要干嘛？
  - 操作register的值
- Register在哪里
  - Register File
- 怎么算结果？结果存在哪儿？
  - ALU计算，结果返回Register File

I-Type Instructions



I-type

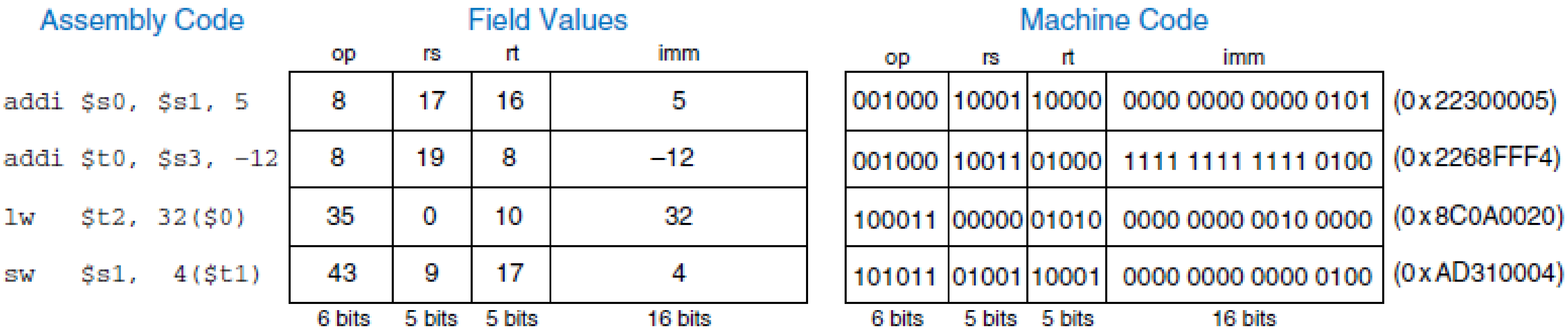
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

- Immediate-type: I-type instructions use two register operands and one immediate operand

- The first three fields, *op*, *rs*, *and rt*, are like those of R-type instructions. The *imm* field holds the 16-bit immediate.

- The operation is determined solely by the **opcode**

Example



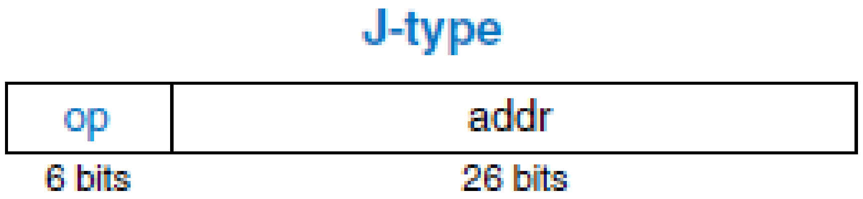
Assembly Code	Field Values				Machine Code				
	op	rs	rt	imm	op	rs	rt	imm	
`addi $s0, $s1, 5`	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
`addi $t0, $s3, -12`	8	19	8	–12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
`lw   $t2, 32($0)`	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
`sw   $s1,  4($t1)`	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

- **Note position of rs and rt**!!!!!!
- Sign extension for I-type: since the *imm field* is 16 bits but used in 32 bit operations. Computer offset by sign extension.

I-type instruction datapath

- Instruction 从哪里来？
  - Instruction Memory
- Instruction 要干嘛？
  - 操作register的值,
  - Instruction里16bit的imm field 需要 sign ext
- Register在哪里
  - Register File
- 怎么算结果？结果存在哪儿？
  - ALU计算，结果返回Register File

J-Type Instruction



J-type

op	addr
6 bits	26 bits

- Jump type:
  - This instruction format uses a single **26-bit address operand, *addr***.
  - J-type instructions begin with a **6-bit opcode**.
  - The remaining bits are used to specify an address, *addr*.

- Only 2 type j-type instruction
  - J : Jump
  - Jal : Jump and link

J-type continue

- How does 26 bit coded address field specify a target in jump in 32 bit structure?

- Destination address(32 bit in total) = {PC[31:28], the 26 bits, 00}

- 为什么最后两位是固定的00？

- 因为PC+4 for each instruction, ------100 = 4

J-type instruction datapath

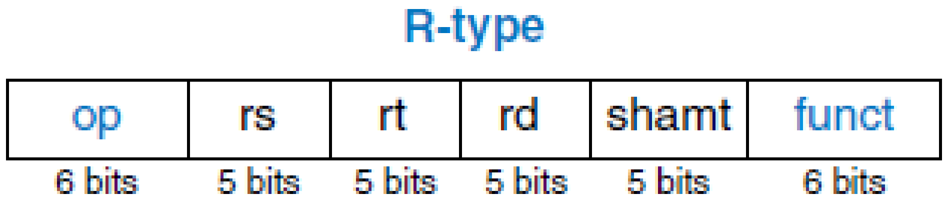
- Instruction 从哪里来？
    - Instruction Memory
- Instruction 要干嘛？
    - 操作PC的值
    - 修改PC，覆盖原有PC
- 怎么算结果？结果存在哪儿？
    - ALU计算，结果返回PC

考点

- 翻译Instructions：
  - FROM Assembly to binary codes
  - From binary codes to Assembly
- Understanding datapath
  - Given instruction, draw the path
  - Given instruction, write the control signals

Example(2012 Fall) 3’~4’ per question

- write the equivalent machine code instruction in the space provided or write the equivalent assembly
- 1. addu $t2, $t0, $t1
- 步骤：
  - 判断是R, I, J哪种type ---- R type
  - 查表
    - addu 100001 $d, $s, $t
    - Opcode = 000000(6位帝皇丸)
    - Rs = $t0 = 8 = 01000
    - Rt = $t1 = 9 = 01001
    - Rd = $t2 = 10 = 01010
    - shamt = xxxxx
    - Funct =
  - Answer = 000000 01000 01001 01010 100001

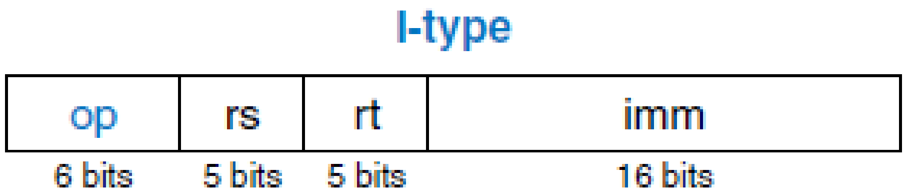


R-type

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

Example(2012 Fall) 3’~4’ per question

- 2. lw $t0, 20($s0)



I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

- I – type
  - lw 100011 $t, i ($s)
  - Rs = $s0 = 10000
  - Rt = $t0 = 01000
  - Imm = 20 = 000……010100

Answer = 100011 10000 01000 000…010100

Example(2012 Fall) 3’~4’ per question

- For the following machine code instructions, provide the equivalent assembly language instruction in the space provided.

a) 001110 01000 00010 0000000011111111

步骤

1. 看前六位判断是否为R type，若不是000000则直接查表
2. 若不是J 和 JAL 则一定是I type，则确定好划分格式为 6(op) 5(s) 5(t) 16(imm)
3. 查表对照

Xori $v0, $t0, 255

帮助记忆的小东东

- Sorted
    - **S** 在前 **T** 在后， **D**estination 在最后
    - R – op6, s5, t5, d5, shamt5, funct6
    - I – op6, s5, t5, imm16
    - J – op6, addr26

一点小东东

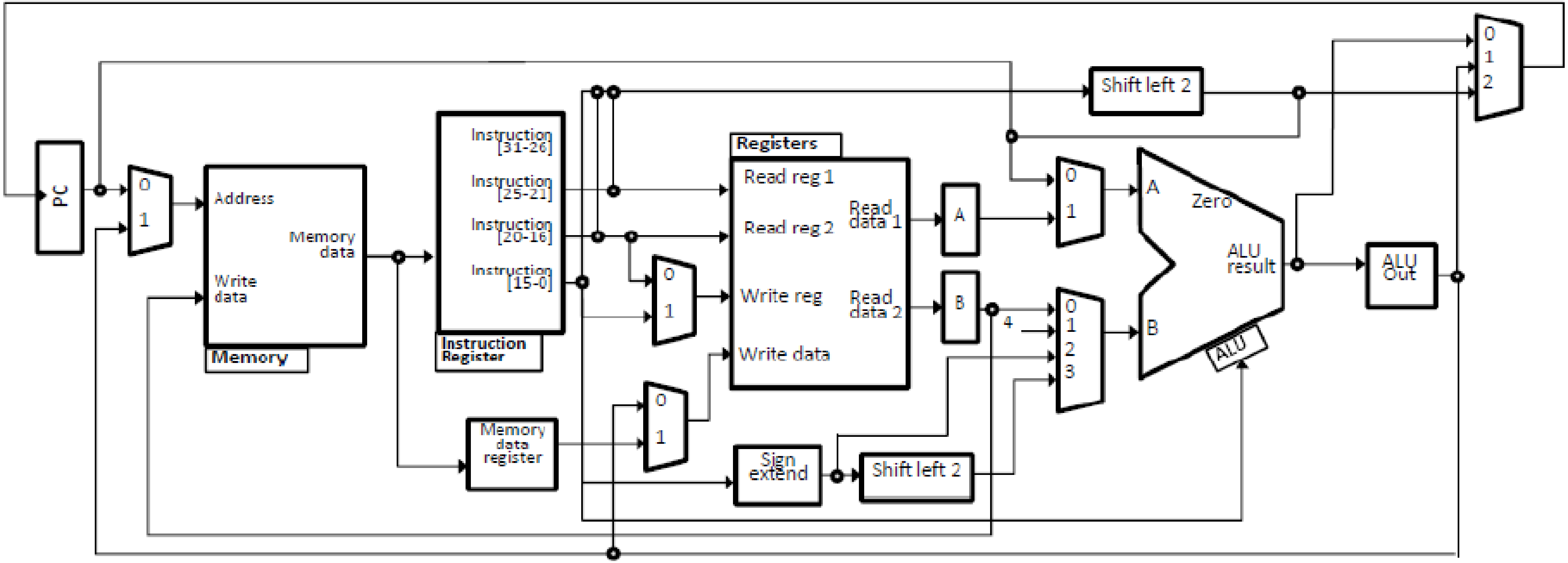
- ***MULT – Multiply*** **（*R - type*）**
  - Multiplies $s by $t and stores the result in $LO.
  - Syntax： mult $s, $t

- **MFLO --** ***Move from LO*** **（*R-type*）**
  - The contents of register LO are moved to the specified register.
  - Syntax：mflo $d
  - Encoding：000000 00000 00000 ddddd 00000 010010
- Similarly for division
  - Search about : DIV and MFHI

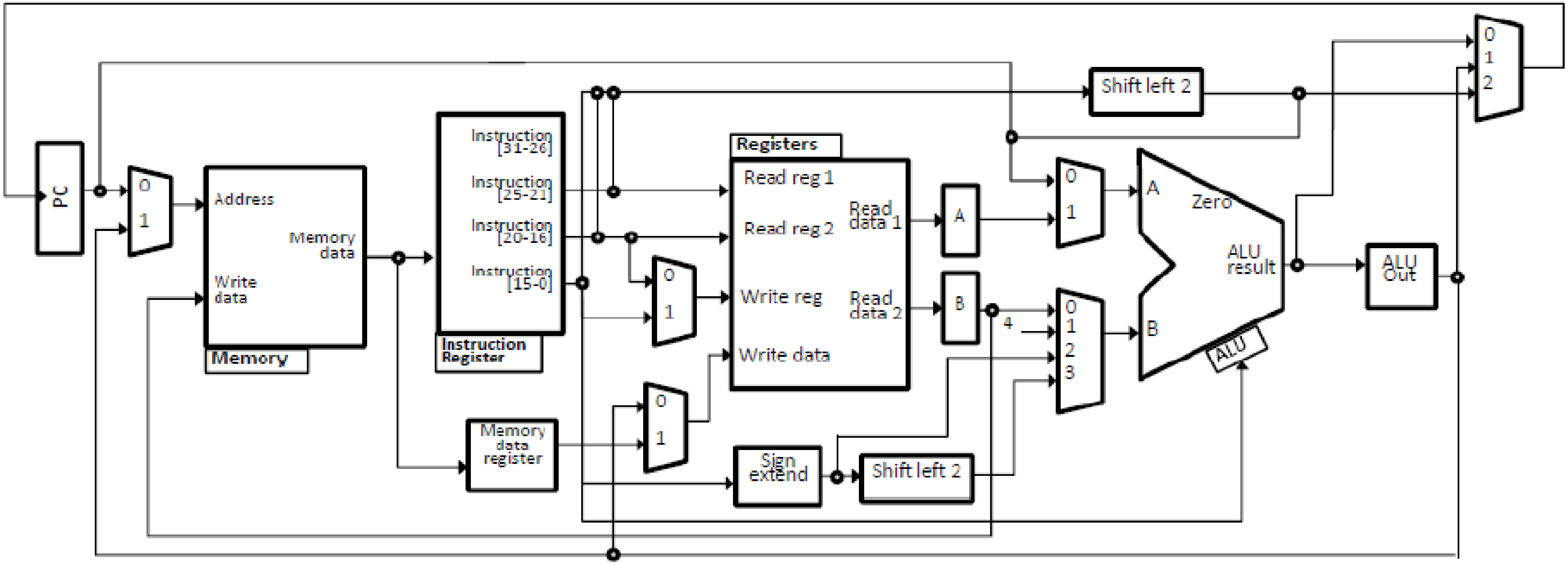
考点2 datapath

- 解题步骤

1. **起点** 和 **终点 (R?I?J?)**
2. 找到datapath
3. 推导信号
   a) 先从 Read/Write 等enable 信号开始，datapath所经之处，都是1，不需要的都是0（不是1就是0）
   b) 填剩下的MUX signal，datapath所经之处，按照选择填信号，没经过的mux信号都是don’t cares
   c) RegDst rule: high for 3 register operation;
   Low for 2 register operation, X if not using register file
4. Finished


PC
0
1
Address
Memory data
Write data
Memory
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction Register
Memory data register
0
1
0
1
Registers
Read reg 1
Read reg 2
Write reg
Write data
Read data 1
Read data 2
Sign extend
Shift left 2
A
B
0
1
0
1
2
3
4
A
B
Zero
ALU result
ALU
ALU Out
Shift left 2
0
1
2

Example： Reduce PC by value stored in $t0

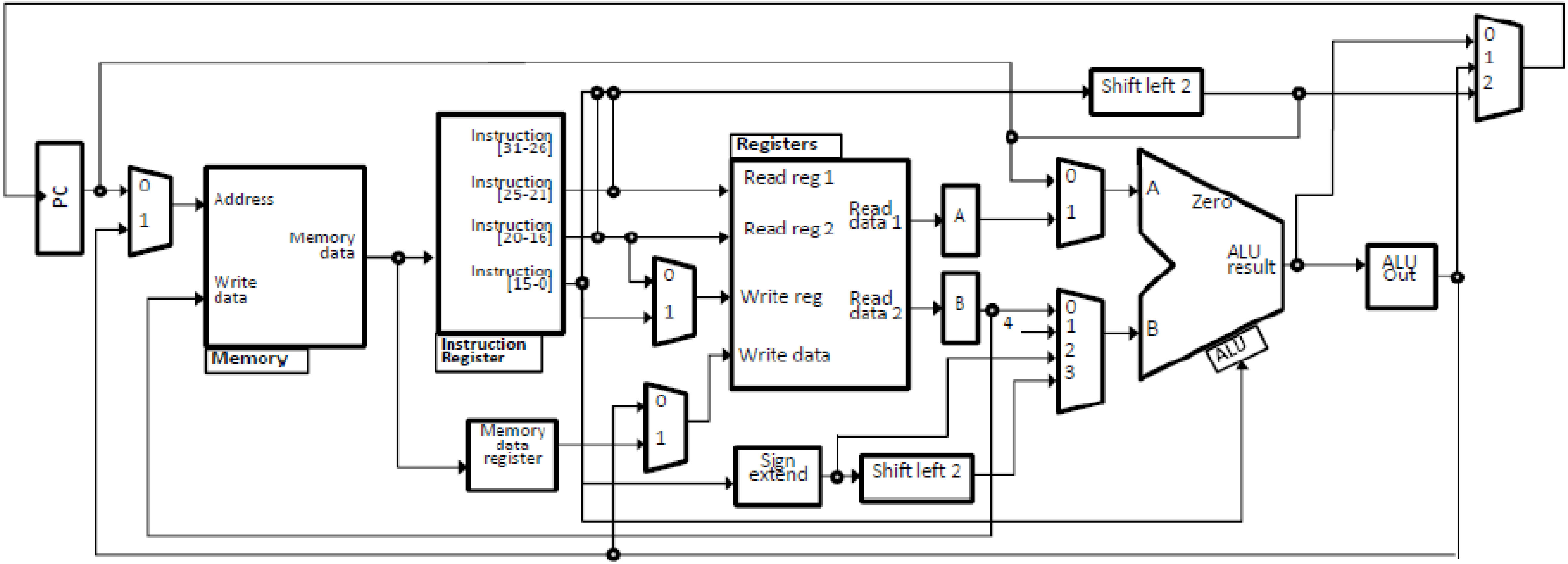


考点：Control signals

- PCWrite = PCWriteCond =
- MemRead = MemWrite =
- IRWrite = RegWrite =

- IorD = MemToReg =
- PCSource = ALUOp =
- ALUSrcA = ALUSrcB =

Example： Add 64 to $s0 and store the result back in $s0



考点：Control signals

- PCWrite = PCWriteCond =
- MemRead = MemWrite =
- IRWrite = RegWrite =

- IorD = MemToReg =
- PCSource = ALUOp =
- ALUSrcA = ALUSrcB =

Welcome to the very end of CSC258…

Assembly Code sectioning syntax

- .data

Indicates the start of the data declarations.

- .text

Indicates the start of the program instructions.

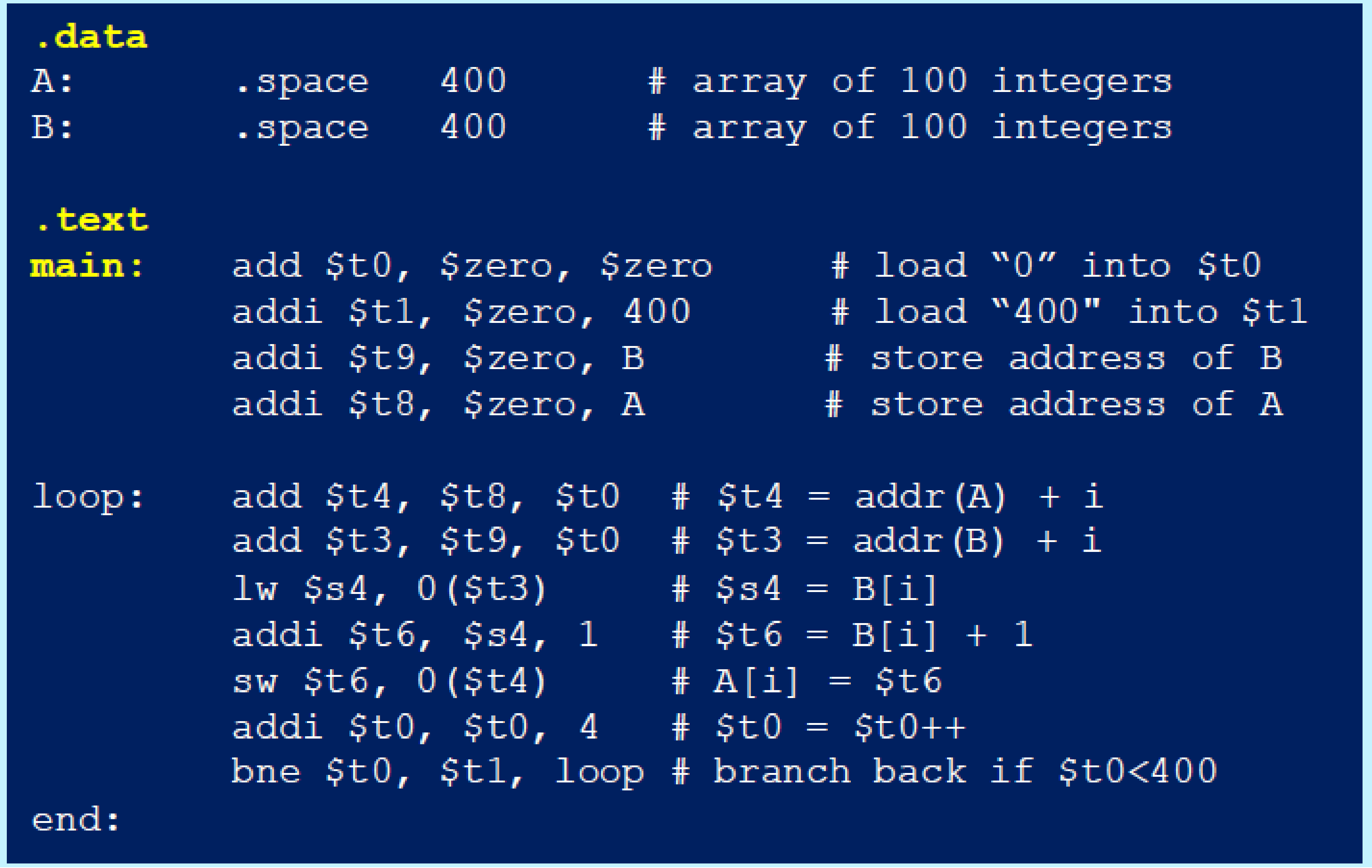
- main:

The initial line to run when executing the program.

Very like C

A Example From Course

- 遍历数组A和B
- A中元素的值全都赋值为B[i] + 1
- $t3, $t4 存地址（pointer）
- $s4, $t6存值



```
.data
A:        .space   400       # array of 100 integers
B:        .space   400       # array of 100 integers

.text
main:     add $t0, $zero, $zero     # load “0” into $t0
          addi $t1, $zero, 400      # load “400" into $t1
          addi $t9, $zero, B        # store address of B
          addi $t8, $zero, A        # store address of A

loop:     add $t4, $t8, $t0   # $t4 = addr(A) + i
          add $t3, $t9, $t0   # $t3 = addr(B) + i
          lw $s4, 0($t3)      # $s4 = B[i]
          addi $t6, $s4, 1    # $t6 = B[i] + 1
          sw $t6, 0($t4)      # A[i] = $t6
          addi $t0, $t0, 4    # $t0 = $t0++
          bne $t0, $t1, loop # branch back if $t0<400
end:
```

Next Week

- Assembly Programs
  - Loops
  - Branches
  - Recursion and stack

  - review