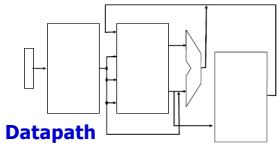
Lecture 11: Single Cycle Processor

Topics

Design a single cycle processor: step-by-step

- Requirements of the instruction set
- Assembling a datapath using
  - Functional units
  - Other combinational and sequential elements
- Designing and assembling control



Instruction Set and Functional Units

Computer Architecture = Instruction Set Architecture + Machine Organisation

Instruction set architecture defines:	Organisation defines:
Organisation of Programmable Storage	Capabilities & Performance
Data Types & Data Structures: Encodings & Representations	Characteristics of Principal Functional Units (e.g., Registers, ALU, Logic Units, ...)
Instruction Set: Instruction Formats Modes of Addressing i.e. Accessing Data Items and Instructions Exceptional Conditions	Interconnection of components Information flow between components Means by which such information flow is controlled

How to Design a Processor: step-by-step

1. Analyse instruction set → datapath requirements
   - the meaning of each instruction is given by the register transfers (we use Verilog notation for RTL - Register Transfer Language)
   - datapath must provide storage
   - and datapath must support transfer
2. Select a set of datapath components and establish clocking methodology
3. Assemble datapath to meet the requirements
4. Analyse implementation of each instruction
   - to determine setting of control points that trigger the register
5. Assemble the control logic

Logical Register Transfers

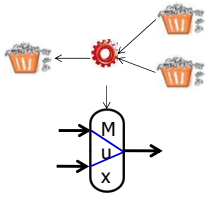
inst	Register Transfers	
ADD	R[rd] ⇐ R[rs] + R[rt]	PC ⇐ PC + 4
SUB	___	
OR	___	
SLT	if (R[rs] < R[rt]) then R[rd] ⇐ 1 else R[rd] ⇐ 0	PC ⇐ PC + 4
LOAD	R[rt] ⇐ MEM[R[rs] + sign_ext(Imm16)]	PC ⇐ PC + 4
STORE	___	
BEQ	if (R[rs] == R[rt]) then PC ⇐ (PC + 4) + (sign_ext(Imm16) || 00) else PC ⇐ PC + 4	
JUMP	PC ⇐ (PC+4)[31,28] || target || 00	

Step 1: Requirements of the Instruction Set

- Memory
  - instruction & data (has to be separate in a single cycle datapath, can't have two reads from one memory)
- Registers
  - (32 x 32) - read **rs**, read **rt**, write **rt** or **rd**
- PC
  - add 4
  - add 4 and extended immediate (shifted left by 2)
  - add 4 and concatenate PC[31,28] || target || 00
- Sign Extender
  - To extend a 16-bit field to 32 bits
- Add, Sub, AND, OR operation (ALU)
  - two registers or
  - extended immediate and register
- Compare two registers
  - also a function of ALU

Step 2: Components of the Datapath

- Storage Elements
  - register file (32 registers with 32 bits each)
  - PC (32-bit register)
  - instruction memory (32-bit words)
  - data memory (32-bit words)
- Combinational Elements
  - Adders
  - ALU
  - sign extender
  - MUX's
- Clocking methodology for storage (sequential) elements

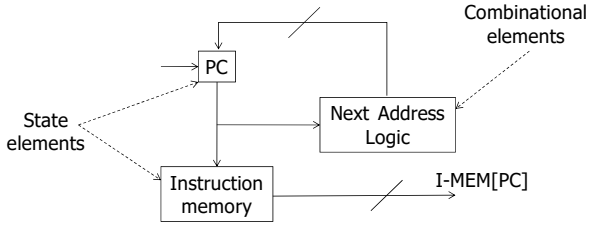


Step 3: Assembling datapath

- Register Transfer Requirements for each instruction → Datapath Assembly
  - have to provide paths to enable transfers
- For example:
  - PC ⇐ (PC + 4) + (sign_ext(Imm16) || 00)
  - needs a path from sign extender to PC and a path from output of PC to input of PC
- For running instructions
  - Instruction Fetch and Read Operands
    - Are the steps the same for each instruction?
  - Execute Operation and Store results
    - Are the steps the same for each instruction?

3a: Overview of the Instruction Fetch Unit

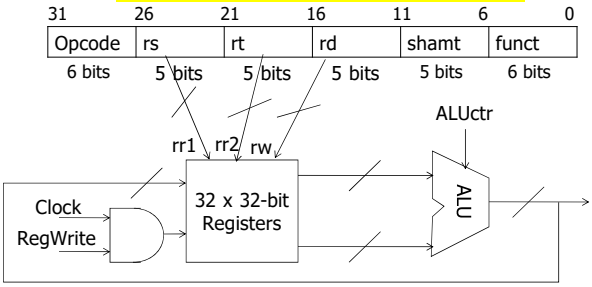
- The common register transfer operations
  - Fetch the instruction using the Program Counter (PC) at the beginning of an instruction's execution
  - Update the program counter at the end of the instruction's execution



3b: Add, Subtract, logical, SLT

- R[rd] ⇐ R[rs] op R[rt] Example: add rd, rs, rt

Instruction from instruction memory - Mem[PC]



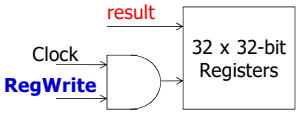
- MARS X-Ray shows the execution process dynamically.
- Logisim or logisim-evolution is also a good software to visualise execution stages over datapath.
- The MIPS simulator at https://www3.ntu.edu.sg/home/smitha/FYP_Gerald/index.html is also interesting.

RegWrite

- This control signal is asserted for all instructions which write to a register
  - all arithmetic and logical, load instruction
- It has to be negated (no write to register) for all others
  - branch on equal, jump, store instruction

RegWrite ⇐ if ((op== Store) || (op==BEQ) || (op==JUMP)) then 0 else 1

- RegWrite AND Clock
  - Register is only written on clock edge



ALUctr

- The ALU control signal selects the operation in the ALU
- ALU 3-bit control input

```
000 AND
001 OR
010 ADD
110 SUB
111 SLT  (set-on-less-than)
```

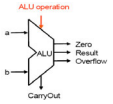
- For LW (load) and SW (store) ALU is used to calculate the address
  - What operation is used?

  R[rt] ⇐ MEM[R[rs] + sign_ext(Imm16)]
- On BEQ (branch on equal) ALU is used to perform subtraction
  - why subtraction?

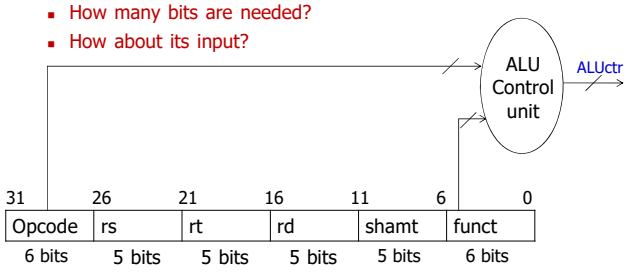
  a = b means (a – b) = 0

ALU Control Unit

- The ALU operation is determined by:



- The ALU Control Unit
  - Its output is: ALUctr control signal
  - How many bits are needed?
  - How about its input?



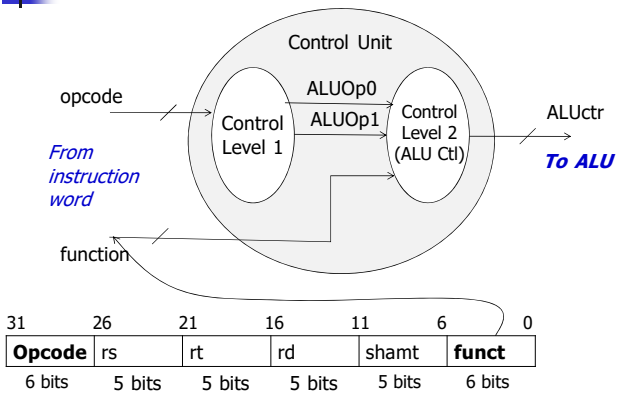
Computing **ALUctr**

- Multiple level of decoding
  - two bit ALUOp determined from opcode (bits 31-26):

    00 = lw, sw - always add
    01 = beq - always subtract
    10 = R-type - funct (bits 5-0) used to define operation

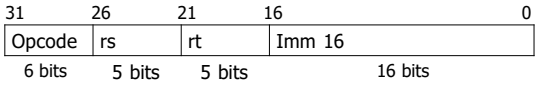
ALUOp		Funct field						Operation code (ALUctr)	Operation name	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0			
0	0	x	x	x	x	x	x	010		lw/sw
0	1	x	x	x	x	x	x	110		beq
1	x	x	x	0	0	0	0	010		R-Type
1	x	x	x	0	0	1	0	110		
1	x	x	x	0	1	0	0	000		
1	x	x	x	0	1	0	1	001		
1	x	x	x	1	0	1	0	111		

ALU Control Unit



3c: Load Operation

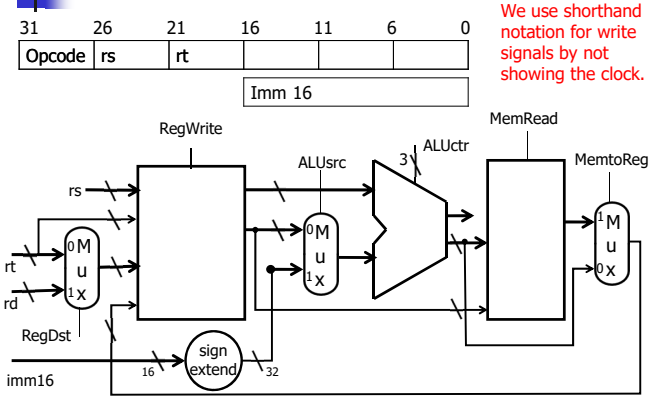
Two bit ALUOp: **00** = lw, sw - always be translated into 'add'



Instruction:	**lw** rt, rs, imm16
Address:	R[rs] + SignExt[imm16]
Register transfers:	R[rt] ⇐ Mem[R[rs] + SignExt[imm16]]

ALUOp		**Funct** field						Operation code (**ALUctr**)	Operation name
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	**0**	x	x	x	x	x	x	**010**	[ADD]

Datapath for Load Operation

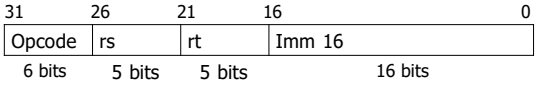


Control signals for lw

- New signals
  - RegDst ⇐ (op == R-Type) ? 1 : 0
    - for load, destination register is **rt**
    - unlike in R-type instructions
    - the MUX is used to select
  - ALUsrc ⇐ (op==R-Type || op==BEQ) ? 0 : 1
    - for load, the source of the second ALU input is sign-extended immediate value
    - the MUX to select between the above and readRegister2
  - MemRead ⇐ (op == Load) ? 1 : 0
    - the ALU output is used as an address for data memory
  - MemtoReg ⇐ (op == Load) ? 1 : 0
    - selects memory as the source for register write
    - unlike all R-type instructions
    - The MUX to select between memory and ALU output
- Also needed
  - RegWrite ⇐ ((op== Store) || (op==BEQ) || (op==JUMP)) ? 0 : 1
    - Same as for R-type instructions
  - ALUctr
    - Same as for R-type instructions
  - All other Write control signals deasserted

3d: Store Operation

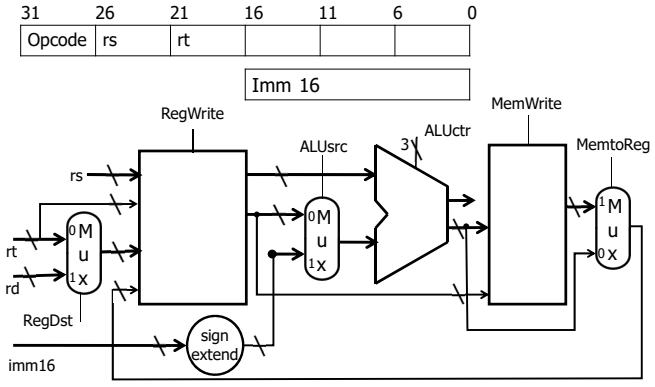
Two bit ALUOp: **00** = lw, sw - always 'add'



Instruction:	**sw** rt, rs, imm16
Address:	R[rs] + SignExt[imm16]
Register transfers:	Mem[R[rs] + SignExt[imm16]] ⇐ R[rt]

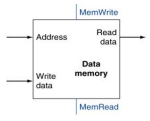
ALUOp		**Funct** field						Operation code (normalized)	Operation name
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	**0**	x	x	x	x	x	x	**010**	[ADD]

Datapath for Store Operation



Control signals for sw

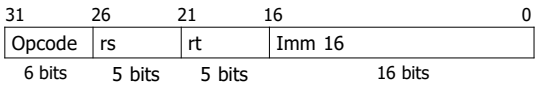
- New signals **MemWrite** ⇐ (op == Store)
  - if asserted
    - the content of register **rt** is written to memory at the address calculated by the ALU
  - if not asserted (as in ALL other instructions)
    - the content of memory does not change
  - MemWrite AND Clock



Why do we need MemWrite?

- Also needed
  - ALUctr
    - Same as for R-type instructions
  - ALUsrc
    - Same as for lw to calculate the address
  - All other Write control signals deasserted

3e: The Branch Instruction



Instruction: **beq** rs, rt, imm16

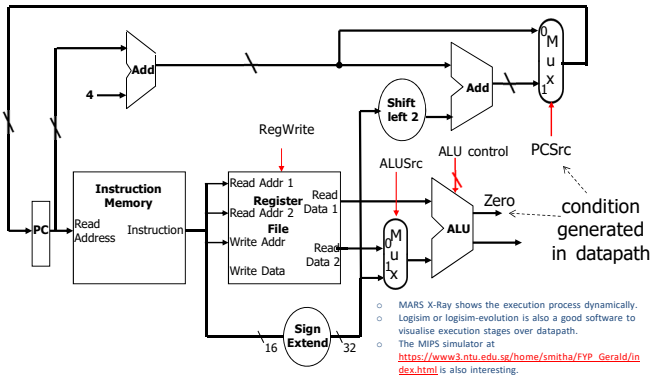
Operation:

```
Zero ⇐ R[rs] == R[rt]

if (Zero)
    PC ⇐ PC + 4 + ( SignExt(imm16) << 2 )
else
    PC ⇐ PC + 4
```


Datapath for Branch Operations

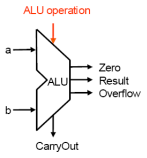
beq rs, rt, imm16 Datapath generates condition (zero)



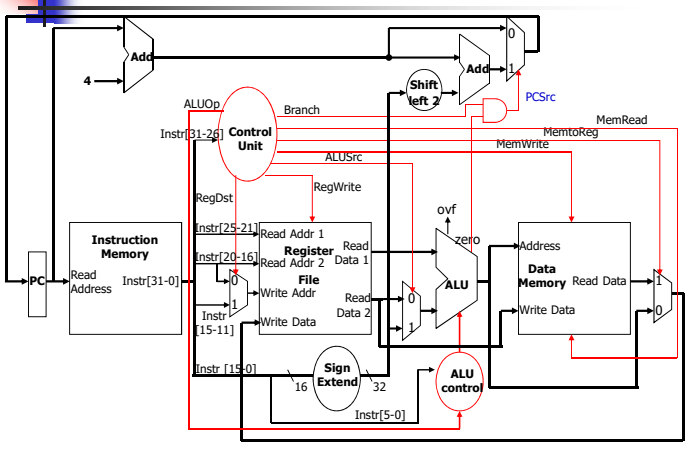
- MARS X-Ray shows the execution process dynamically.
- Logisim or logisim-evolution is also a good software to visualise execution stages over datapath.
- The MIPS simulator at https://www3.ntu.edu.sg/home/smitha/FYP_Gerald/index.html is also interesting.

Control signals for branch

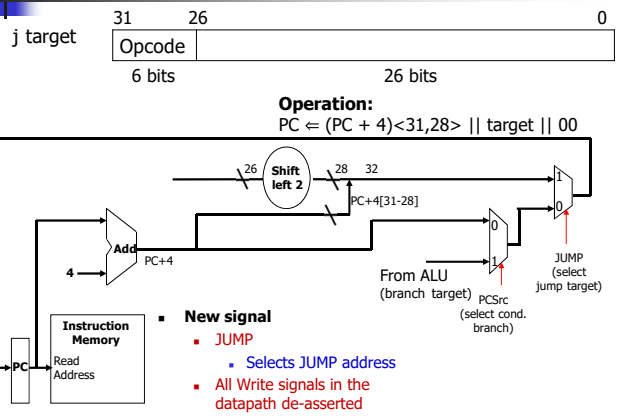
- New signals
  - Branch ⇐ (op == BEQ)
    - So we know it is branch instruction. Once ALU operation is complete it will be used to produce PCsrc
  - Zero
    - Produced by ALU if ALU result is zero ie. Registers are equal when subtracted
  - PCsrc
    - PCsrc := Zero AND Branch meaning that PCsrc is asserted if it is a Branch instruction and ALU result is zero (Zero = 1 signal is true)
- Also needed
  - ALUctr
    - Same as for R-type instructions to enforce subtraction
  - All Write control signals deasserted



Putting it All Together: A Single Cycle Datapath



What about JUMP?



Operation:
PC ⇐ (PC + 4)<31,28> || target || 00

- **New signal**
  - JUMP
    - Selects JUMP address
  - All Write signals in the datapath de-asserted

Step 5: Logic for each control signal

Jump ⇐ (op == Jump)
PCsrc ⇐ if (op == BEQ) then subject to Zero else 0
Branch ⇐ (op == BEQ)
PCsrc ⇐ Branch AND Zero

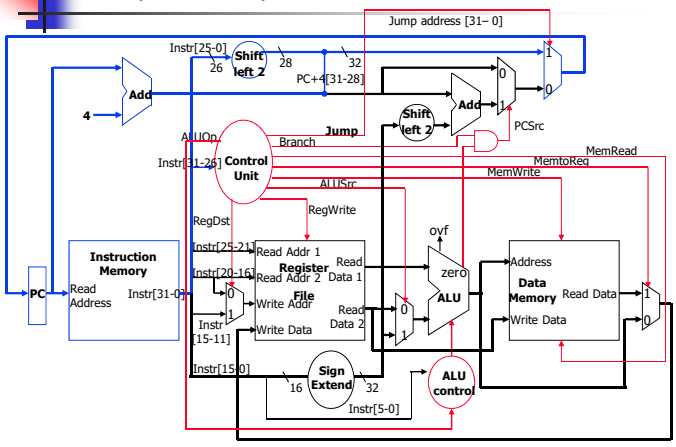
Zero is the signal from ALU

ALUsrc ⇐ if (op==R-Type || op==BEQ) then 0 ("reg2") else 1 ("imm32")
ALUctr ⇐ if (op == R-Type) then funct
elseif (op == BEQ) then "sub" else "add"

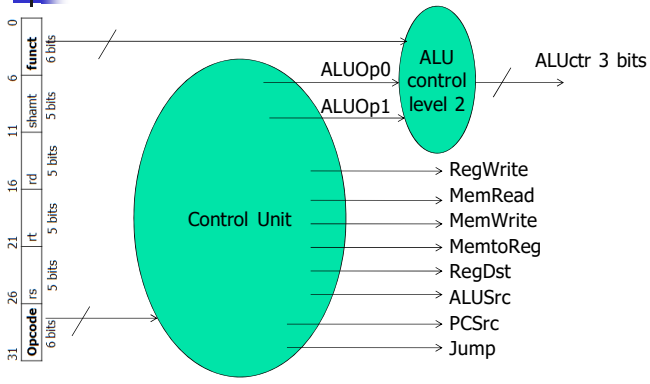
MemRead ⇐ (op == Load)
MemWrite ⇐ (op == Store)
MemtoReg ⇐ (op == Load)
RegWrite ⇐ ((op== Store) || (op==BEQ) || (op==JUMP)) ? 0 : 1
RegDst ⇐ (op == R-Type)

Complete datapath

Refer to FIGURE 4.24



Complete control unit



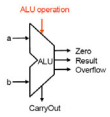
Complete control unit

Opcode	rs	rt	rd	shamt	funct

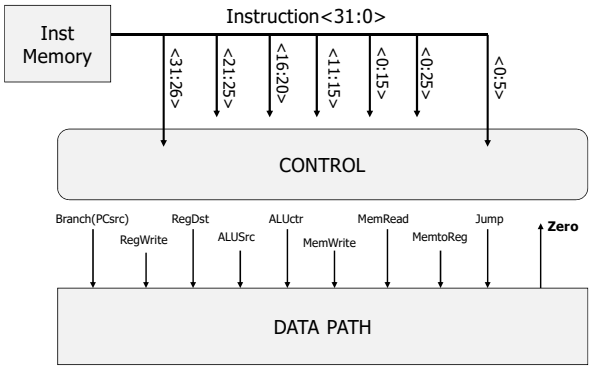
OP (5:0)	INSTRUCTION	CONTROL OUTPUTS: REGDST	ALUSRC	MEMTOREG	REGWRITE	MEMREAD	MEMWRITE	BRANCH	JUMP	ALUOP(1:0)
000000	R-type	1	0	0	1	0	0	0	0	10
001000	Addi	0	1	0	1	0	0	0	0	00
100011	Lw	0	1	1	1	1	0	0	0	00
101011	Sw	X	1	X	0	0	1	0	0	00
000100	Beq	X	0	X	0	0	0	1	0	01
000010	j	X	X	X	0	0	0	0	1	00

The controller must also generate four ALUCONTROL signals using the FUNCT input and ALUOP output:

ALUOP(1:0)	FUNCT(3:0)	ALUCONTROL(3:0)
00	XXXX	0010
01	XXXX	0110
1X	0000	0010
1X	0010	0110
1X	0100	0000
1X	0101	0001
1X	1010	0111



Step 4: Given Datapath: RTL → Control



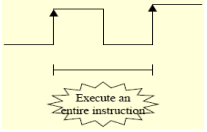
Control signal table

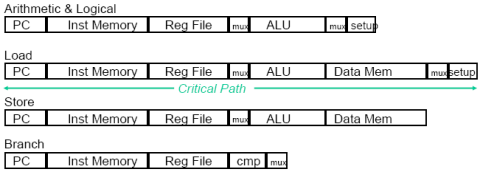
Operation	RegDst	RegWrite	ALUSrc	ALUctr	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUctr for R-type instructions depends on the instructions' func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq ***and*** the ALU's Zero output is true.

Drawback of this Single Cycle Processor

- Execution Time = Insts * CPI * Cycle Time
- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Single cycle processor:
  - Advantage: CPI = 1
  - Disadvantage: long cycle time





Drawback of this Single Cycle Processor

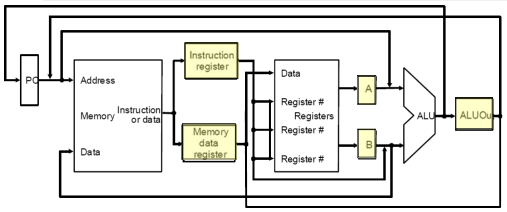
- Long cycle time:
  - Cycle time must be long enough for **WHICH? instruction:**

    Instruction Memory Access Time +
    Register File Access Time +
    ALU Delay (address calculation) +
    Data Memory Access Time +
    Register File Setup Time +
    Clock Skew (*)

    (*) difference in time for clock signal to reach state elements
- If the cycle time is long enough for the longest instruction, it is long enough for all other instructions
  - All instructions take as much time as the slowest.

Solution to single cycle problems

- Single Cycle Problems:
  - What if we had a more complicated instruction like floating point operation?
- Possible Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath in Pipelining

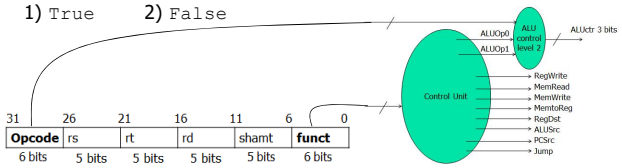


Additional "internal" registers

Revision and quiz

- It's important to understand an instruction in its mnemonic format, encoding format, register transfer representation, and corresponding datapath assembling. Use Load operation as an example to illustrate your understanding.
- The following drawing is correct.

  1) `True` 2) `False`



- The sw instruction doesn't write any registers.

  1) `True` 2) `False`

Recommended readings

General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials

PH6, §4.1-§4.4, P256-P284: The processor
PH5, §4.1-§4.4, P244-P272: The processor
PH4, §4.1-§4.4, P300-P329: The processor

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)
https://www.elsevier.com/books-and-journals/book-companion/9780128201091

PH5: companion materials (e.g. online sections for further readings)
http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263