



**Processor: Datapath**

© IT - TDT Computer Organisation

# Acknowledgement

* The contents of these slides have origin from School of Computing, National University of Singapore.
* We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

© IT - TDT Computer Organisation

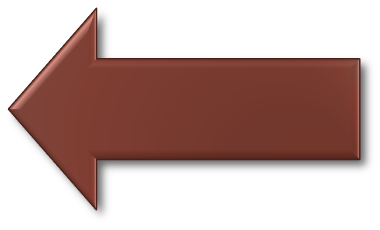
# Policies for students

* These contents are only used for students PERSONALLY.
* Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

# Road Map: **Part II**

|  |
| --- |
| n **Processor Datapath** q Generic Execution  Stages q MIPS Execution Stages q Constructing Datapath |

**Performance**



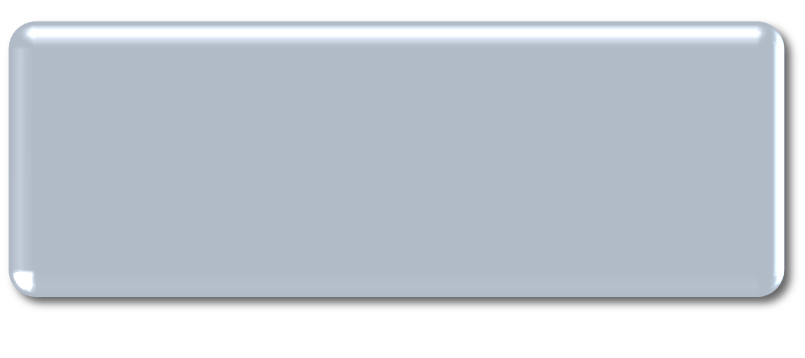
**Assembly**

**Language**



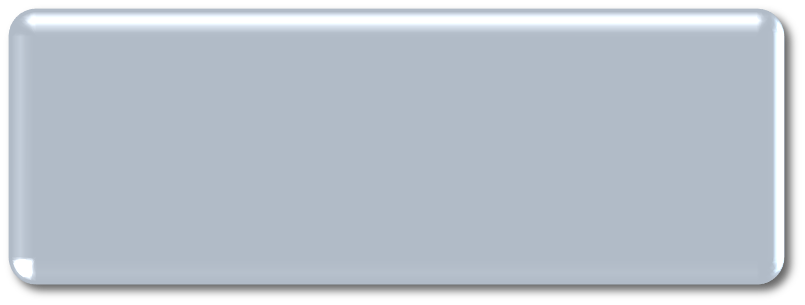
**Processor:**

**Datapath**

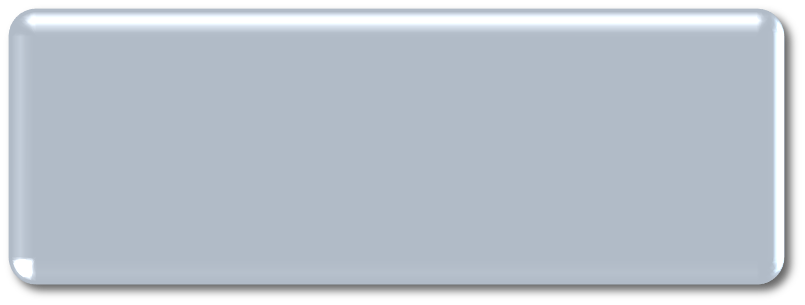


**Processor:**

**Control**



**Pipelining**



**Cache**

Building a Processor: Datapath & Control

* Two major components for a processor:
* **Datapath**
* Collection of components that process data
* Performs the arithmetic, logical and memory operations
* **Control**
* Tells the datapath, memory, and I/O devices what to do according to program instructions

# MIPS Processor: Implementation

* Simplest possible implementation of a subset of the core MIPS ISA:
* **Arithmetic and Logical operations**
* **add**, **sub**, **and**, **or**, **addi**, **andi**, **ori**, **slt**
* **Data transfer instructions**
* **lw, sw**
* **Branches**
* **beq, bne**
* Shift instructions (**sll**, **srl**) and J-type instructions (**j**) will not be discussed here
* Left as exercises J

# Recap: Instruction Execution Cycle



***Instruction***

***Fetch***

***Instruction***

***Decode***

***Operand***

***Fetch***

***Execute***

***Result***

***Write***

**Next Instruction**

* **Fetch:**
* Get instruction from memory
* Address is in **P**rogram **C**ounter (PC) Register
* **Decode:**
* Find out the operation required
* **Operand Fetch:**
* Get operand(s) needed for operation
* **Execute:**
* Perform the required operation • **Result Write (WriteBack):**
* Store the result of the operation

# MIPS Instruction Executions

* Show the actual steps for 3 representative MIPS instructions
* Fetch and Decode stages not shown: • The standard steps are performed

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **lwlw$3$3,, 2020( ($1 ))** | **beqbeq$1$1, , $2$2, , labelofst** |
| **add $3, $1, $2** |
|  |
| **Fetch** | ***standard*** | ***standard*** | ***standard*** |
| **Decode** |
| **Operand Fetch** | o Read [**$1**] as *opr1* o Read [**$2**] as *opr2* | o Read [**$1**] as *opr1* o Use ***20*** as *opr2* | o Read [**$1**] as *opr1* o Read [**$2**] as *opr2* |
| **Execute** | *Result* = *opr1* + *opr2* | o *MemAddr* = opr1 + opr2 o Use *MemAddr* to read data from memory | *Taken* = (*opr1* == *opr2* )?  *Target* = (**PC**+4) or  (**PC**+4) + **ofst** |
| **Result Write** | *Result* stored in **$3** | *Memory* data stored in **$3** | **PC** = *Target* |

* **opr** = Operand n **ofst = offset**
* **MemAddr** = Memory Address

# 5-STAGE MIPS EXECUTION

* Design changes:
* Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
* Split *Execute* into ***ALU*** (Calculation) and ***Memory Access***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | **lwlw$3$3,, 2020( ($1 $1))** | **beqbeq$1$1, , $2$2, , labelofst** |
| **addadd $3$3, , $1$1, , $2$2** |  |
|  |
| **Fetch** | Read inst. at [**PC**] |  | Read inst. at [**PC**] | Read inst. at [**PC**] |
| **Decode** &  **Operand Fetch** | o Read [**$1**] as *opr1* o Read [**$2**] as *opr2* |  | o Read [**$1**] as *opr1* o Use ***20*** as *opr2* | o Read [**$1**] as *opr1* o Read [**$2**] as *opr2* |
| **ALU** | *Result* = *opr1* + *opr2* |  | *MemAddr* = opr1 + opr2 | *Taken* = (*opr1* == *opr2* )?  *Target* = (**PC**+4) or  (**PC**+4) + **ofst** |
| **Memory Access** |  |  | Use *MemAddr* to read datefrom memory |  |
| **Result Write** | *Result* stored in **$3** |  | *Memory* data stored in **$3** | **PC** = *Target* |

Let’s Build a MIPS Processor!

* What we are going to do:
* Look at each stage closely, figure out the requirements and processes
* Sketch a high level block diagram, then zoom in for each elements
* With the simple starting design, check whether different type of instructions can be handled:
* Add modifications when needed è Study the design from the viewpoint of a designer, instead of a “tourist” J

# **Fetch Stage**: Requirements

* Instruction **Fetch Stage**:

1. Use the **P**rogram **C**ounter (**PC**) to fetch the instruction from **memory**

* PC is implemented as a special register in the processor

**2. Increment** the PC by 4 to get the address of the next instruction:

* How do we know the next instruction is at PC+4?
* Note the exception when branch/jump instruction is executed
* Output to the next stage (**Decode**):
* The instruction to be executed

**Fetch Stage**

:

Block Diagram

**Add**

**PC**

**4**

**Read**

**address**

**Instruction**

***Instruction***

***memory***

A register

Memory which

stores

program

instructions

A simple

adder

**Decode Stage**

**Instruction**

# Element: **Instruction Memory**

* Storage element for the instructions
* Recall: **sequential circuit**

***Instruction***

***Memory***

**Instruction**

**Address**

**Instruction**

* Has an internal state that stores information
* Clock signal is assumed and not shown
* Supply instructions given the address
* Given instruction address M as input, the **Memory** memory outputs the content at address M **………..**
* Conceptual diagram of the memory layout is **2048 add $3, $1, $2** given on the right è **2052 sll $4, $3, 2**

**2056 andi $1, $4, 0xF**

**…… ………..**

## Element: **Adder**

* Combinational logic to implement the addition of two numbers
* **Inputs:**

**Sum**

***Add***

**A**

**B**

**A+B**

* Two 32-bit numbers **A**, **B** • **Output:**
* Sum of the input numbers, **A + B**
* Just a 32-bit version of the adder discussed in first part of the course J

## The Idea of Clocking

* It seems that we are reading and updating the PC at the same time: • How can it works properly?
* **Magic of clock**:
* PC is read during the first half of the clock period and it is updated with PC+4 at the **next rising clock edge**

**Add**

**PC**

**4**

**Read**

**address**

**Instruction**

***Instruction***

***memory***

**In**

Clk

Time

**PC**

**100**

**104**

**108**

**112**

**In**

**104**

**108**

**112**

**116**

## **Decode Stage**: Requirement

* Instruction **Decode Stage**:
* Gather data from the instruction fields:
  1. Read the **opcode** to determine instruction type and field lengths
  2. Read data from all necessary registers
* Can be two (e.g. **add**), one (e.g. **addi**) or zero (e.g. **j**)
* Input from previous stage (**Fetch**):
* Instruction to be executed
* Output to the next stage (**ALU**):
* Operation and the necessary operands

## **Decode Stage**: Block Diagram

**Fetch Stage**

Inst.

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Read**

**data 1**

**Read**

**data 2**

**Data**

**Register**

**Number**

***Register***

***File***

**5**

**5**

**5**

**Execute Stage**

**Operands**

Collection of

registers, known

as

**register file**

# Element: **Register File**

* A collection of 32 registers:
* Each is 32-bit wide and can be read/written by specifying register number
* Read at most two registers per instruction
* Write at most one register per instruction
* **RegWrite** is a control signal to indicate:
* Writing of register
* 1(True) = Write, 0 (False) = No Write

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

**Data**

**Data**

**Register**

**Number**

***Register***

***File***

**5**

**5**

**5**

**32**

**32**

**RegWrite**

## Decode Stage: **R-Type Instruction**

**add $8, $9, $10**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**rd**

**15:11**

**shamt**

**10:6**

**funct**

**5:0**

**000000**

**01001**

**01010**

**01000**

**00000**

**100000**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

**RegWrite**

***Register***

***File***

**5**

**5**

**5**

content of

register

**$9**

content of

register

**$10**

Result to be

stored

(

produced by later

stage)

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**Notation:**

Inst [Y:X]

=

bits X to Y in Instruction

**32**

**32**

## Decode Stage: **I-Type Instruction**

**addi $21, $22, -50**

**001000**

**10110**

**10101**

**1111 1111 1100 1110**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

**RegWrite**

***Register***

***File***

**5**

**5**

**5**

Content of

register

**$22**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**Problems:**

-

Destination

**$21**

is in the "wrong position"

-

**Read Data 2**

is an immediate value, not from register

**32**

**32**

Decode Stage: **Choice in Destination addi $21, $22, -50**

**001000**

**10110**

**10101**

**1111 1111 1100 1110**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

***Register***

***File***

**5**

**5**

**5**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**RegDst**

**:**

A control signal to choose

either Inst[20:16] or Inst[15:11]

as the write register number

**Solution (**

Wr

. Reg. No.

**):**

Use a

**multiplexer**

to choose the

correct write register number

based on instruction type

**32**

**32**

## Recap: **Multiplexer**

* **Function:**
* Selects one input from multiple input lines Control

in

0

in

n

-

1

out

m

**.**

**.**

**.**

**M**

**U**

**X**

* **Inputs:** • ***n*** lines of same width
* **Control:** • ***m*** bits where n = 2*m*
* **Output:**
* Select ith input line if control=i Control=0 à select in0

Control=3 à select in3

## Decode Stage: **Choice in Data 2 addi $21, $22, -50**

**001000**

**10110**

**10101**

**1111 1111 1100 1110**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

**5**

**5**

**5**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**ALUSrc**

**RegWrite**

**Sign**

**Extend**

**16**

**32**

***Register***

***File***

**ALUSrc**

**:**

A control signal to

choose either

"Read data 2" or

the sign extended

Inst[15:0] as the

second operand

**Solution (**

Rd. Data 2

**):**

Use a

**multiplexer**

to choose the correct operand 2.

Sign extend the 16

-

bit immediate value to 32

-

bit

**32**

**32**

Decode Stage: **Load Word Instruction**

* Try it out: "**lw $21, -50($22)**"
* Do we need any modification?

**100011**

**10110**

**10101**

**1111 1111 1100 1110**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

**5**

**5**

**5**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**ALUSrc**

**RegWrite**

Sign

Extend

**16**

**32**

***Register***

***File***

**32**

**32**

## Decode Stage: **Branch Instruction**

* Example: "**beq $9, $0, 3**"

•

We will tackle this problem in the ALU Stage

J

**000100**

**01001**

**00000**

**0000 0000 0000 0011**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

**5**

**5**

**5**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**ALUSrc**

**RegWrite**

Sign

Extend

**16**

**32**

***Register***

***File***

**32**

**32**

* Need to calculate branch outcome and target at the same time!

## **Decode Stage:** Summary

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

***Registers***

**5**

**5**

**5**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**ALUSrc**

**RegWrite**

Sign

Extend

**16**

**32**

Operand 1

Operand 2

Inst[31:0]

**32**

**32**

### ALU Stage: Requirement

* Instruction **ALU Stage**:
* ALU = Arithmetic-Logic Unit
* Perform the real work for most instructions here
* Arithmetic (e.g. **add**, **sub**), Shifting (e.g. **sll**), Logical (e.g. **and**, **or**)
* Memory operation (e.g. **lw**, **sw**): Address calculation
* Branch operation (e.g. **bne**, **beq**): Perform register comparison and target address calculation
* Input from previous stage (**Decode**):
* Operation and Operand(s)
* Output to the next stage (**Memory**):
* Calculation result

### ALU Stage: Block Diagram

**ALU**

**result**

***ALU***

**Decode Stage**

Operands

**Memory Stage**

Logic to perform

arithmetic and

logical operations

### Element: Arithmetic Logical Unit

* **ALU (Arithmetic-logical unit**)

**ALU**

**result**

***ALU***

**ALUcontrol**

**4**

**isZero**

**?**

**A**

**B**

**A op B**

**(**

**A op B) == 0?**

* Combinational logic to implement arithmetic and logical operations
* **Inputs:**
* Two 32-bit numbers

|  |  |
| --- | --- |
| **ALUcontrol** | **Function** |
| **0000** | **AND** |
| **0001** | **OR** |
| **0010** | **add** |
| **0110** | **subtract** |
| **0111** | **slt** |
| **1100** | **NOR** |

* **Control:**
* 4-bit to decide the particular operation
* **Outputs:**
* Result of arithmetic/logical operation
* A 1-bit signal to indicate whether result is zero

ALU Stage: **Non-Branch Instructions**

* We can handle non-branch instructions easily:

#### **add $8, $9, $10**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

***Register***

***File***

**5**

**5**

**5**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**ALUSrc**

**RegWrite**

Sign

Extend

**16**

**32**

**ALU**

**result**

***ALU***

**ALUcontrol**

**4**

**isZero**

**?**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**rd**

**15:11**

**shamt**

**10:6**

**funct**

**5:0**

**000000**

**01001**

**01010**

**01000**

**00000**

**100000**

**ALUcontrol**

**:**

Set using

opcode

+

funct

field (more in next lecture)

## ALU Stage: **Brach Instructions**

* Branch instruction is harder as we need to perform two calculations:
* Example: "**beq $9, $0, 3**"

**1. Branch Outcome:**

* Use ALU unit to compare the register
* check (how?)

**2. Branch Target Address:**

* + Introduce additional logic to calculate the address
  + Need **PC** (from Fetch Stage)
  + Need **Offset** (from Decode Stage)

Complete ALU Stage

**000100**

**01001**

**00000**

**0000 0000 0000 0011**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Write**

**data**

**Read**

**data 1**

**Read**

**data 2**

***Register***

***File***

**5**

**5**

**5**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**ALUSrc**

**RegWrite**

Sign

Extend

**16**

**32**

**ALU**

**result**

***ALU***

**ALUcontrol**

**4**

**isZero**

**?**

**Left Shift**

**2**

**-**

**bit**

**PC**

***Add***

**4**

***Add***

**M**

**U**

**X**

**PCSrc**

**PCSrc**

**:**

Control Signal

to select

between

PC+4) or

(

Branch Target

E.g. "**beq $9, $0, 3**"

### Memory Stage: Requirement

* Instruction **Memory Access Stage**:
* Only the load and store instructions need to perform operation in this stage:
* Use memory address calculated by ALU Stage
* Read from or write to data memory
* All other instructions remain idle
* Result from ALU Stage will pass through to be used in Result Store (Writeback) stage if applicable
* Input from previous stage (**ALU**):
* Computation result to be used as memory address (if applicable) • Output to the next stage (**Writeback**):
* Result to be stored (if applicable)

### Memory Stage: Block Diagram

**ALU Stage**

Result

**Result Store**

**Stage**

Memory which

stores data

values

***Data***

***Memory***

**Address**

**Read**

**Data**

**Write**

**Data**

**MemRead**

**MemWrite**

### Element: Data Memory

* Storage element for the data of a program
* **Inputs: MemWrite**
* Memory Address

***Data***

***Memory***

**Address**

**Read**

**Data**

**Write**

**Data**

* Data to be written (Write Data) for store instructions
* **Control:**
* Read and Write controls; only one can be asserted at any point of time
* **Output: MemRead**
* Data read from memory (Read Data) for load instructions

### Memory Stage: Load Instructions

• Only relevant parts of Decode & ALU Stages are shown

#### **lw $21, -50($22)**

**000100**

**01001**

**00000**

**0000 0000 0000 0011**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**ALUSrc**

**RR1**

**RR2**

**WR**

**WD**

**RD1**

**RD2**

***Registers***

**5**

**5**

**5**

**RegWrite**

Sign

Extend

**16**

**32**

**ALU**

**result**

***ALU***

**ALUcontrol**

**4**

**100011**

**10110**

**10101**

**1111 1111 1100 1110**

**Address**

**Write**

**Data**

**MemRead**

**MemWrite**

***Data***

***Memory***

**Read**

**Data**

### Memory Stage: Store Instructions

• Need ***Read Data 2*** (Decode) as the ***Write Data***

#### **sw $21, -50($22)**

**000100**

**01001**

**00000**

**0000 0000 0000 0011**

**Immediate**

**15:0**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**RR1**

**RR2**

**WR**

**WD**

**RD1**

**RD2**

***Registers***

**5**

**5**

**5**

**RegWrite**

Sign

Extend

**16**

**32**

**ALU**

**result**

***ALU***

**ALUcontrol**

**4**

**101011**

**10110**

**10101**

**1111 1111 1100 1110**

**Address**

**Write**

**Data**

**MemRead**

**MemWrite**

***Data***

***Memory***

**Read**

**Data**

#### Memory Stage: **Non-Memory Instructions**

• Add a multiplexer to choose the result to be stored

##### add $8, $9, $10

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**RegDst**

**Inst [**

**15:0]**

**M**

**U**

**X**

**RR1**

**RR2**

**WR**

**WD**

**RD1**

**RD2**

***Registers***

**5**

**5**

**5**

**RegWrite**

Sign

Extend

**16**

**32**

**ALU**

**result**

***ALU***

**ALUcontrol**

**4**

***Data***

***Memory***

**Address**

**Read**

**Data**

**Write**

**Data**

**MemWrite**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**rd**

**15:11**

**shamt**

**10:6**

**funct**

**5:0**

**000000**

**01001**

**01010**

**01000**

**00000**

**100000**

**M**

**U**

**X**

**MemToReg**

**MemToReg**

**:**

A control signal to indicate

whether result came from

memory or ALU unit

### Result Write Stage: Requirement

* Instruction **Register Write Stage**:
* Most instructions write the result of some computation into a register
* Examples: arithmetic, logical, shifts, loads, set-less-than • Need destination register number and computation result • Exceptions are stores, branches, jumps:
* There are no results to be written èThese instructions remain idle in this stage
* Input from previous stage (**Memory**):
* Computation result either from memory or ALU

### Result Write Stage: Block Diagram

**Memory Stage**

Result

**Read**

**register 1**

**Read**

**register 2**

**Write**

**register**

**Read**

**data 1**

**Read**

**data 2**

***Registers***

**5**

**5**

**5**

**Write**

**data**

* Result Write stage has no additional element:
* Basically just route the correct result into register file
* The ***Write Register*** number is generated way back in the **Decode** Stage

FetchProcessor: DecodeDatapath

### Result Write Stage: Routing add $8, $9, $10

**Inst [25:21]**

**Inst [20:16]**

**Inst [15:11]**

**M**

**U**

**X**

**Inst [**

**15:0]**

**M**

**U**

**X**

**RR1**

**RR2**

**WR**

**WD**

**RD1**

**RD2**

***Registers***

**5**

**5**

**5**

**RegWrite**

Sign

Extend

**ALU**

**result**

***ALU***

**ALUcontrol**

**4**

***Data***

***Memory***

**Address**

**Read**

**Data**

**Write**

**Data**

**MemWrite**

**opcode**

**31:26**

**rs**

**25:21**

**rt**

**20:16**

**rd**

**15:11**

**shamt**

**10:6**

**funct**

**5:0**

**000000**

**01001**

**01010**

**01000**

**00000**

**100000**

**M**

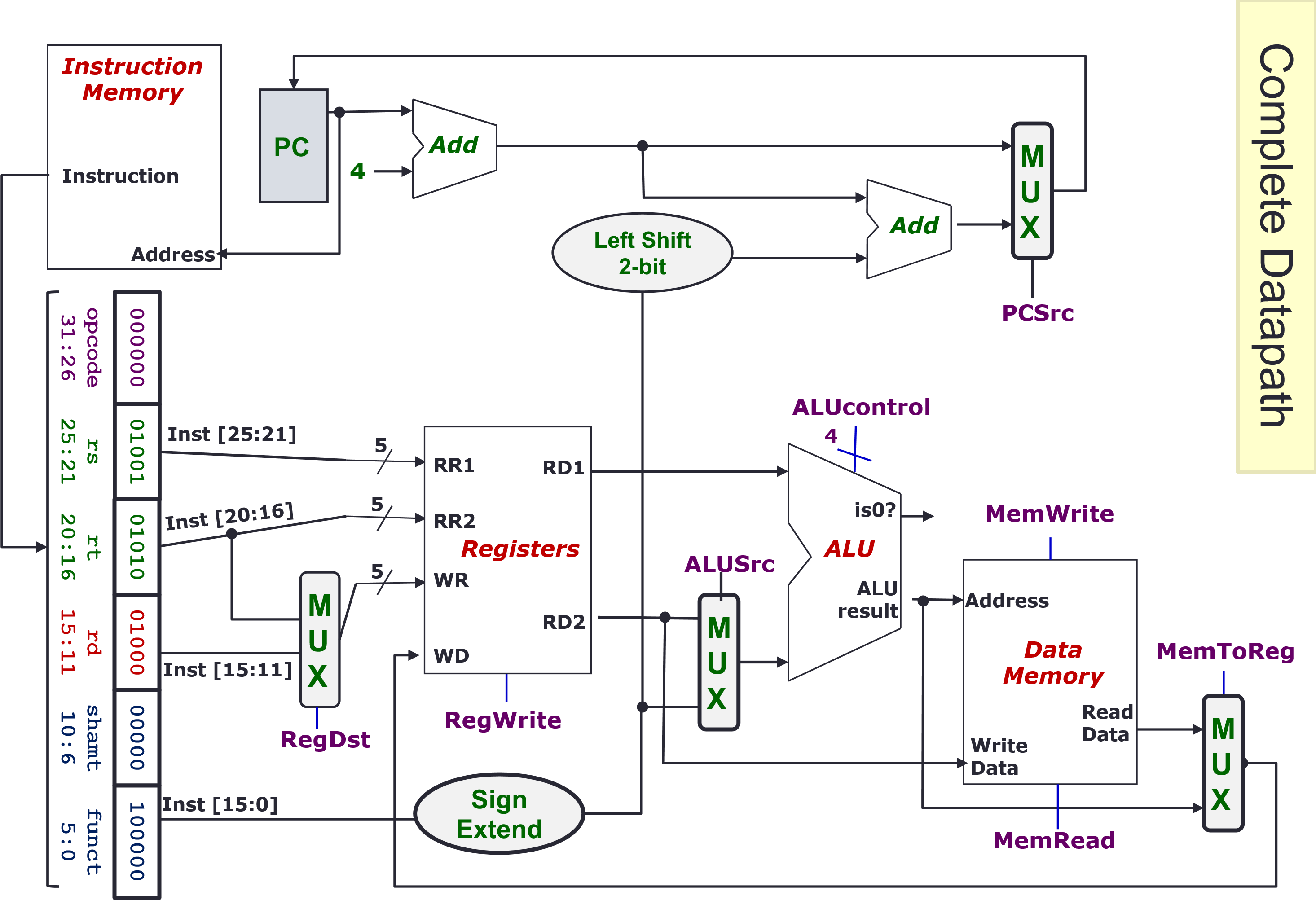
**U**

**X**

**MemToReg**

The Complete Datapath!

* We have just finished “designing” the datapath for a subset of MIPS instructions:
* Shifting and Jumping are not supported
* Check your understanding:
* Take the complete datapath and play the role of controller:
* See how supported instructions are executed
* Figure out the correct control signals for the datapath elements
* Coming up next: **Control** (Lecture #16)



#### Datapath: Generic Steps

PC

instruction

memory

+4

rt

rs

rd

registers

ALU

Data

memory

imm

1

. Instruction

Fetch

2

. Decode/

Register

3

.

ALU

4

. Memory

5

.

Write

Back

Read

Read

Write

Read

#### Datapath Walkthroughs: ADD (1/2)

* **add $r3,$r1,$r2 # r3 = r1+r2**
* Stage 1: Fetch this instruction, increment PC.
* Stage 2: Decode to find that it is an **add** instruction, then read registers **$r1** and **$r2**.
* Stage 3: Add the two values retrieved in stage 2.
* Stage 4: Idle (nothing to write to memory).
* Stage 5: Write result of stage 3 into register **$r3**.

#### Datapath Walkthroughs: ADD (2/2)

**add $r3, $r1, $r2**

PC

instruction

memory

+4

registers

ALU

Data

memory

imm

**2**

**1**

**3**

**add r3, r1, r2**

**reg[1]+reg[2]**

**reg[2]**

**reg[1]**

Read

Read

Write

#### Datapath Walkthroughs: SLTI (1/2)

* **slti $r3,$r1,17**
* Stage 1: Fetch this instruction, increment PC.
* Stage 2: Decode to find it is an **slti**, then read register **$r1**. § Stage 3: Compare value retrieved in stage 2 with the integer 17.
* Stage 4: Go idle.
* Stage 5: Write the result of stage 3 in register **$r3**.

#### Datapath Walkthroughs: SLTI (2/2)

**slti $r3, $r1, 17**

PC

instruction

memory

+4

registers

ALU

Data

memory

imm

**x**

**1**

**3**

**slti r3, r1, 17**

**reg[1]**

**-**

**17**

**17**

**reg[1]**

Read

Read

Write

#### Datapath Walkthroughs: SW (1/2)

* **sw $r3, 20($r1)**
* Stage 1: Fetch this instruction, increment PC.
* Stage 2: Decode to find it is an **sw**, then read registers **$r1** and **$r3**.
* Stage 3: Add 20 to value in register **$r1** (retrieved in stage 2).
* Stage 4: Write value in register **$r3** (retrieved in stage 2) into memory address computed in stage 3.
* Stage 5: Go idle (nothing to write into a register).

#### Datapath Walkthroughs: SW (2/2)

**sw $r3, 20($r1)**

PC

instruction

memory

+4

registers

ALU

Data

memory

imm

**3**

**1**

**x**

**sw r3, 20(r1)**

**reg[1]+20**

**20**

**reg[1]**

**MEM[r1+20]<**

**-**

**r3**

**reg[3]**

Read

Read

Write

Why Five Stages?

* Could we have a different number of stages? § Yes, and other architectures do.
* So why does MIPS have five stages, if instructions tend to go idle for at least one stage?
* There is one instruction that uses all five stages:

the load.

#### Datapath Walkthroughs: LW (1/2)

* **lw $r3, 40($r1)**
* Stage 1: Fetch this instruction, increment PC.
* Stage 2: Decode to find it is a **lw**, then read register **$r1**.
* Stage 3: Add 40 to value in register **$r1** (retrieved in stage 2).
* Stage 4: Read value from memory address compute in stage 3.
* Stage 5: Write value found in stage 4 into register **$r3**.

#### Datapath Walkthroughs: LW (2/2)

**lw $r3, 40($r1)**

PC

instruction

memory

+4

registers

ALU

Data

memory

imm

**x**

**1**

**3**

**lw r3, 40(r1)**

**reg[1]+40**

**r3<**

**-**

**MEM[r1+40]**

**reg[3]**

Read

Read

Write

**40**

**reg[1]**

What Hardware is Needed?

* PC: a register which keeps track of address of the next instruction. § General Purpose Registers
* Used in stage 2 (read registers) and stage 5 (writeback).
* Memory
* Used in stage 1 (fetch) and stage 4 (memory).
* Cache system makes these two stages as fast as the others, on average.

#### Datapath: Summary

* Construct datapath based on register transfers required to perform instructions.
* Control part causes the right transfers to happen.

PC

instruction

memory

+4

rt

rs

rd

registers

ALU

Data

memory

imm

**Controller**

opcode, funct

#### Reading Assignment

* The Processor: Datapath and Control
* 3rd edition: Chapter 5 Sections 5.1 – 5.3
* 4th edition: Chapter 4 Sections 4.1 – 4.3

© IT - TDT Computer Organisation

# Q&A