Midterm Questions Overview

Four questions from the following:

- **Performance Evaluation:** **2 Questions**
  - **Given MIPS code, estimate performance on a given CPU.**
  - **Compare performance of different CPU/compiler changes for a given program. May involve computing execution time, speedup, CPI, MIPS rating, etc.**
  - **Single or multiple enhancement Amdahl's Law given parameters before or after the enhancements are applied.**
- **Adding support for a new instruction to the textbook versions of:** **2 Questions**
  - **Single cycle MIPS CPU**
  - **Multi cycle MIPS CPU**

Dependant RTN for the new instruction and changes to datapath/control.

CPU Organization (Design)

- **Datapath Design:** **Components & their connections needed by ISA instructions**
  - **Capabilities & performance characteristics of principal Functional Units (FUs) needed by ISA instructions**
  - **(e.g., Registers, ALU, Shifters, Logic Units, ...)** ***Components***
  - **Ways in which these components are interconnected (buses connections, multiplexors, etc.).** ***Connections***
  - **How information flows between components.**
- **Control Unit Design:** **Control/sequencing of operations of datapath components to realize ISA instructions**
  - **Logic and means by which such information flow is controlled.**
  - **Control and coordination of FUs operation to realize the targeted Instruction Set Architecture to be implemented (can either be implemented using a finite state machine or a microprogram).**
- **Hardware description with a suitable language, possibly using Register Transfer Notation (RTN).**

ISA = Instruction Set Architecture
The ISA forms an abstraction layer that sets the requirements for both complier and CPU designers

Reduced Instruction Set Computer (RISC)

~1984 ISAs

- Focuses on reducing the number and complexity of instructions of the ISA.

RISC: Simplify ISA → Simplify CPU Design → Better CPU Performance

  - Motivated by simplifying the ISA and its requirements to:

RISC Goals

    - Reduce CPU design complexity
    - Improve CPU performance.

  - <u>CPU Performance Goal:</u> Reduced number of cycles needed per instruction. At least one instruction completed per clock cycle.

- <u>Simplified addressing modes supported.</u>
  - Usually limited to immediate, register indirect, register displacement, indexed.
- <u>Load-Store GPR:</u> Only load and store instructions access memory.
  - (Thus more instructions are usually executed than CISC)
- <u>Fixed-length instruction encoding.</u>
  - (Designed with CPU instruction pipelining in mind).
- Support of delayed branches.
- <u>Examples:</u> MIPS, HP PA-RISC, SPARC, Alpha, POWER, PowerPC.

Chapter 2 (both editions)

RISC ISA Example:

MIPS R3000 (32-bit)

- **Memory: Can address 2^{32} bytes or 2^{30} words (32-bits).**
- **Instruction Categories:**
  - **Load/Store.**
  - **Computational: ALU.**
  - **Jump and Branch.**
  - **Floating Point.**
    - **Using coprocessor**
  - **Memory Management.**
  - **Special.**

5 Addressing Modes:

- **Register direct (arithmetic).**
- **Immedate (arithmetic).**
- **Base register + immediate offset (loads and stores).**
- **PC relative (branches).**
- **Pseudodirect (jumps)**

Operand Sizes:

- **Memory accesses in any multiple between 1 and 4 bytes.**

Registers

R0 - R31

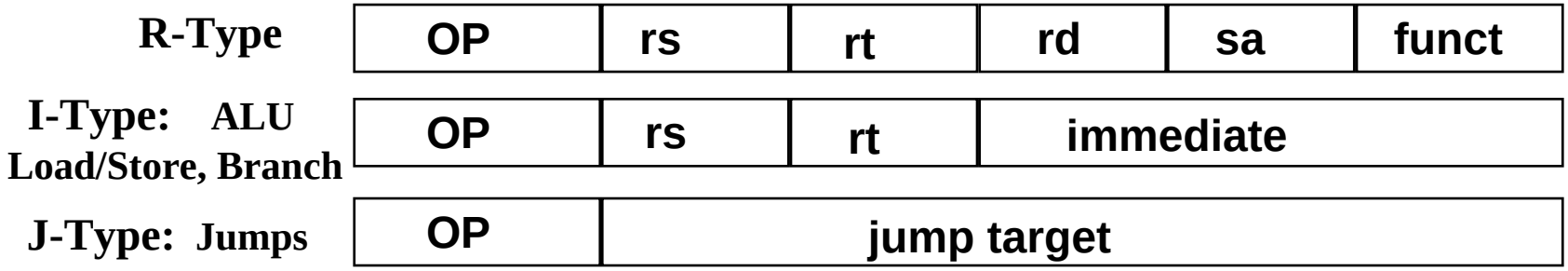
31 GPRs
R0 = 0
(each 32 bits)

PC

HI

LO

Instruction Encoding: 3 Instruction Formats, all 32 bits wide.



Word = 4 bytes = 32 bits

MIPS Register Usage/Naming Conventions

- In addition to the usual naming of registers by $ followed with register number, registers are also named according to MIPS register usage convention as follows:

Register Number	Name	Usage	Preserved on call?
0	$zero	Constant value 0	n.a.
1	$at	Reserved for assembler	no
2-3	$v0-$v1	Values for result and expression evaluation	no
4-7	$a0-$a3	Arguments	yes
8-15	$t0-$t7	Temporaries	no
16-23	$s0-$s7	Saved	yes
24-25	$t8-$t9	More temporaries	no
26-27	$k0-$k1	Reserved for operating system	yes
28	$gp	Global pointer	yes
29	$sp	Stack pointer	yes
30	$fp	Frame pointer	yes
31	$ra	Return address	yes

MIPS Five Addressing Modes

1 **Register Addressing:** e.g. add $1,$2,$3

Where the operand is a register (R-Type)

2 **Immediate Addressing:** e.g. addi $1,$2,100

Where the operand is a constant in the instruction (I-Type, ALU)

3 **Base or Displacement Addressing:** e.g. lw $1, 32($2)

Where the operand is at the memory location whose address is the sum of a register and a constant in the instruction (I-Type, load/store)

4 **PC-Relative Addressing:** e.g. beq $1,$2,100

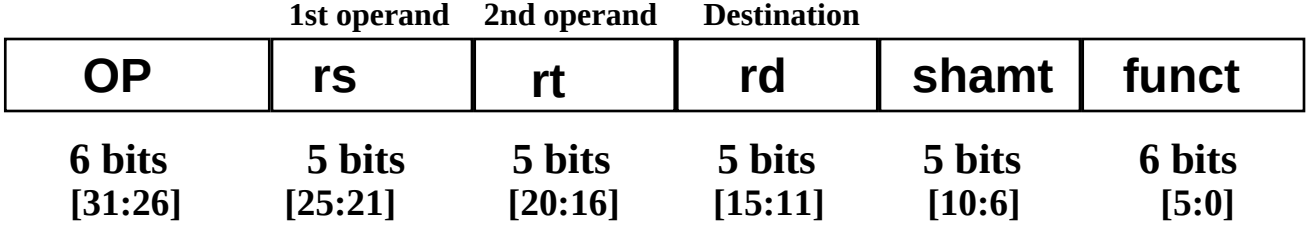
Where the address is the sum of the PC and the 16-address field in the instruction shifted left 2 bits. (I-Type, branches)

5 **Pseudodirect Addressing:** e.g. j 10000

Where the jump address is the 26-bit jump target from the instruction shifted left 2 bits concatenated with the 4 upper bits of the PC (J-Type)

MIPS R-Type (ALU) Instruction Fields

R-Type: All ALU instructions that use three registers



- **op: Opcode, basic operation of the instruction.**
  - **For R-Type op = 0**
- **rs: The first register source operand.**
- **rt: The second register source operand.**
- **rd: The register destination operand.**
- **shamt: Shift amount used in constant shift operations.**
- **funct: Function, selects the specific variant of operation in the op field.**

Rs, rt , rd
are register specifier fields

Independent RTN:

R[rd] ← R[rs] funct R[rt]
PC ← PC + 4

Funct field value examples:
Add = 32 Sub = 34 AND = 36 OR =37 NOR = 39

Operand register in rs

Destination register in rd

Operand register in rt

Examples:

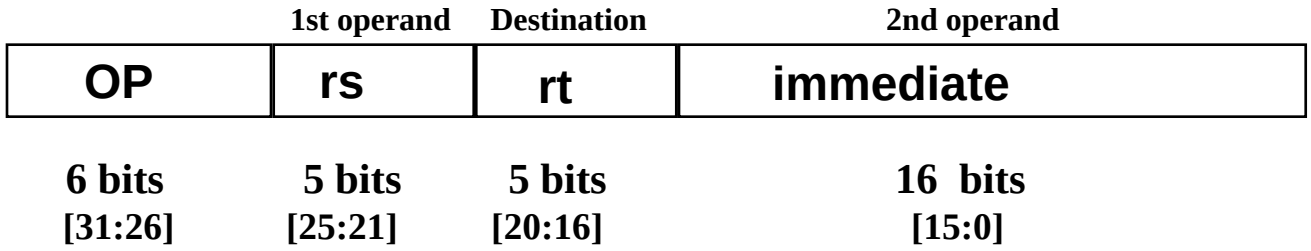
add $1,$2,$3
sub $1,$2,$3

and $1,$2,$3
or $1,$2,$3

R-Type = Register Type
Register Addressing used (Mode 1)

MIPS ALU I-Type Instruction Fields

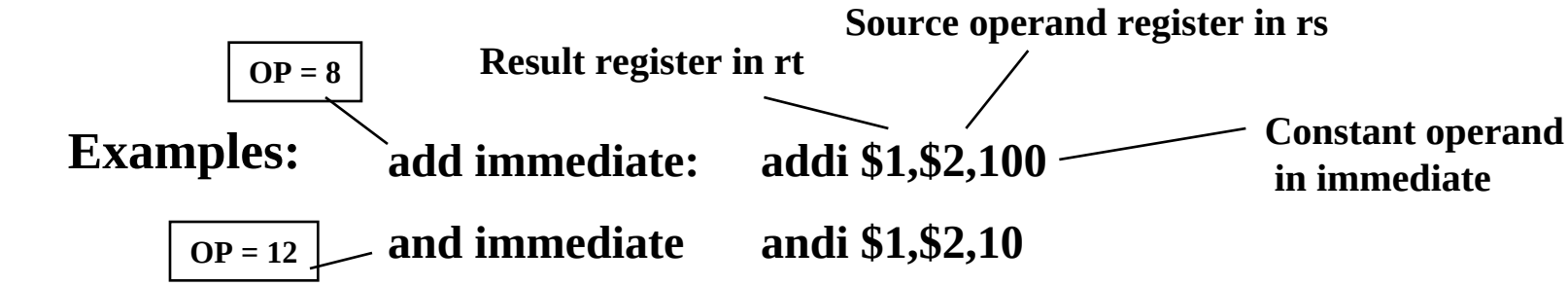
I-Type ALU instructions that use two registers and an immediate value (I-Type is also used for Loads/stores, conditional branches).



- **op: Opcode, operation of the instruction.**
- **rs: The register source operand.**
- **rt: The result destination register.**
- **immediate: Constant second operand for ALU instruction.**

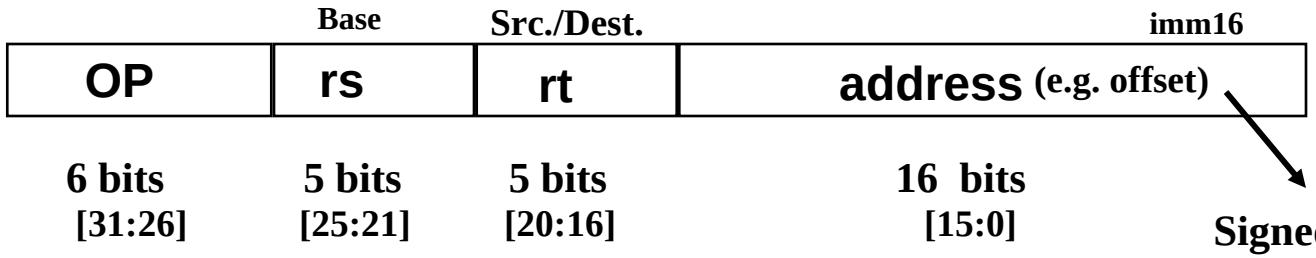
Independent RTN for addi:

R[rt] ← R[rs] + immediate
PC ← PC + 4



I-Type = Immediate Type
Immediate Addressing used (Mode 2)

MIPS Load/Store I-Type Instruction Fields



- **op: Opcode, operation of the instruction.**
  - **For load word op = 35, for store word op = 43.**
- **rs: The register containing memory base address.**
- **rt: For loads, the destination register. For stores, the source register of value to be stored.**
- **address: 16-bit memory address offset in bytes added to base register.**

Examples:

Store word: **sw $3, 500($4)**

source register in rt — Offset — base register in rs

Mem[R[rs] + address] ← R[rt]
PC ← PC + 4

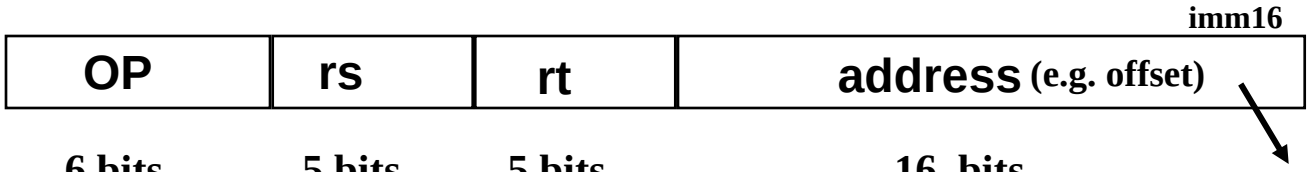
Load word: **lw $1, 32($2)**

Destination register in rt — Offset — base register in rs

R[rt] ← Mem[R[rs] + address]
PC ← PC + 4

Base or Displacement Addressing used (Mode 3)

MIPS Branch I-Type Instruction Fields



- **op: Opcode, operation of the instruction.**
- **rs: The first register being compared**
- **rt: The second register being compared.**
- **address: 16-bit memory address branch target offset in words added to PC to form branch address.**

Examples:

OP = 4	Branch on equal	beq $1,$2,100	Register in rs: $1; Register in rt: $2; offset in bytes equal to instruction address field x 4
OP = 5	Branch on not equal	bne $1,$2,100	

Added to PC+4 to form branch target

Independent RTN for beq:

R[rs] = R[rt] : PC ← PC + 4 + address x 4
R[rs] ≠ R[rt] : PC ← PC + 4

PC-Relative Addressing used (Mode 4)

MIPS J-Type Instruction Fields

J-Type: Include jump j, jump and link jal

6 bits [31:26]

26 bits [25:0]

Jump target in words

Word = 4 bytes

- op: Opcode, operation of the instruction.
  - Jump j op = 2
  - Jump and link jal op = 3
- jump target: jump memory address in words.

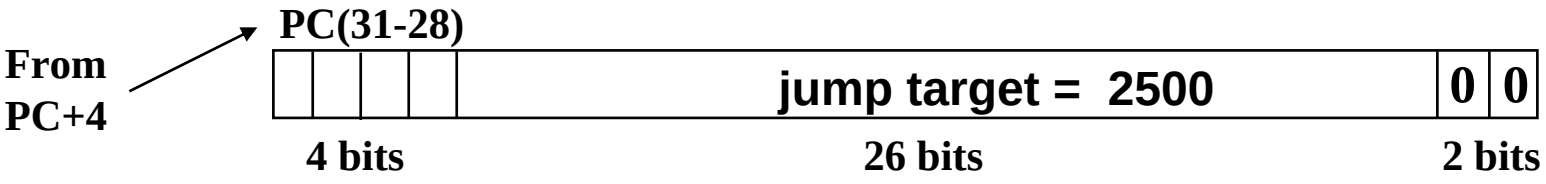
Jump memory address in bytes equal to instruction field jump target x 4

Examples:

Jump j 10000

Jump and link jal 10000

Effective 32-bit jump address: PC(31-28),jump_target,00



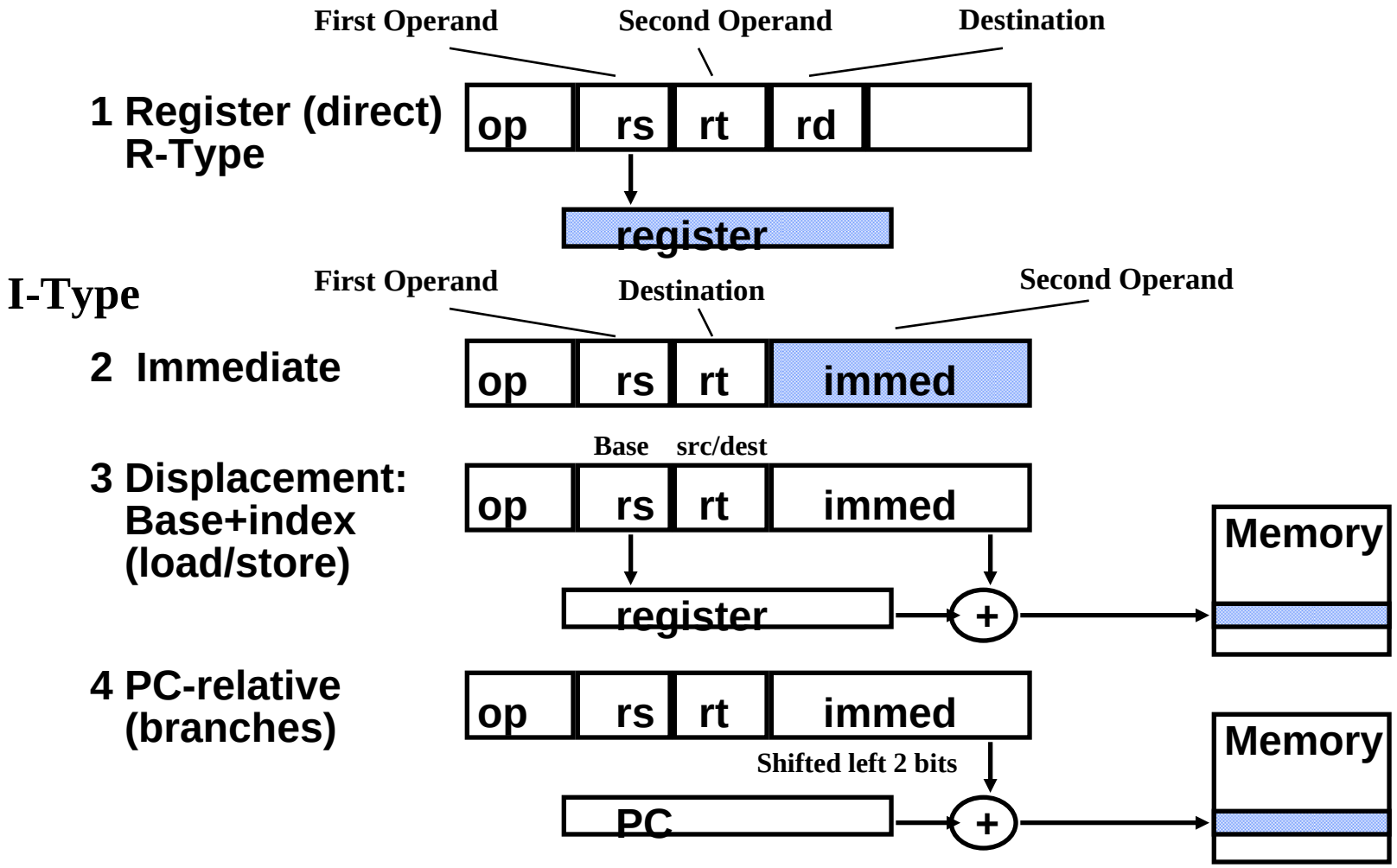
Independent RTN for j:

PC ← PC + 4
PC ← PC(31-28),jump_target,00

J-Type = Jump Type Pseudodirect Addressing used (Mode 5)

MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide



Pseudodirect Addressing (Mode 5) not shown here, illustrated in the last slide for J-Type

MIPS Arithmetic Instructions Examples
(Integer)

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

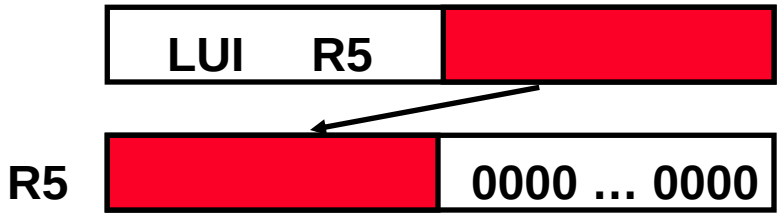
MIPS Logic/Shift Instructions Examples

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

MIPS Data Transfer Instructions Examples

Instruction	*Comment*
sw $3, 500($4)	Store word
sh $3, 502($2)	Store half word
sb $2, 41($3)	Store byte
lw $1, 30($2)	Load word
lh $1, 40($3)	Load half word
lhu $1, 40($3)	Load half word unsigned
lb $1, 40($3)	Load byte
lbu $1, 40($3)	Load byte unsigned
lui $1, 40	Load Upper Immediate (16 bits shifted left by 16)

Word = 4 bytes



MIPS Branch, Compare, Jump Instructions Examples

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100 *Equal test; PC relative branch*
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100 *Not equal test; PC relative branch*
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; 2's comp.*
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; 2's comp.*
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; natural numbers*
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; natural numbers*
jump	j 10000	go to 10000 *Jump to target address*
jump register	jr $31	go to $31 *For switch, procedure return*
jump and link	jal 10000	$31 = PC + 4; go to 10000 *For procedure call*

Example: Simple C Loop to MIPS

- **Simple loop in C:**

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

A[] array of words in memory

- **Assume MIPS register mapping:**

g: $s1, h: $s2, i: $s3, j: $s4, base of A[]: $s5

- **MIPS Instructions:**

```
Loop:   add $t1,$s3,$s3     # $t1= 2*i
        add $t1,$t1,$t1     # $t1= 4*i
        add $t1,$t1,$s5     # $t1=address of A[I]
        lw  $t1,0($t1)      # $t1= A[i]
        add $s1,$s1,$t1     # g = g + A[i]
        add $s3,$s3,$s4     # I = i + j
        bne $s3,$s2,Loop    # goto Loop if i!=h
```

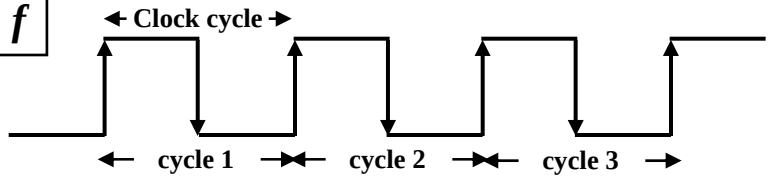
Word = 4 bytes

CPU Performance Evaluation: Cycles Per Instruction (CPI)

- **Most computers run synchronously utilizing a CPU clock running at a constant clock rate:** *Or clock frequency: f*

  where: Clock rate = 1 / clock cycle

  $f = 1 / C$



- **The CPU clock rate depends on the specific CPU organization (design) and hardware implementation technology (VLSI) used.**
- **A computer machine (ISA) instruction is comprised of a number of elementary or micro operations which vary in number and complexity depending on the the instruction and the exact CPU organization (Design).**
  - **A micro operation is an elementary hardware operation that can be performed during one CPU clock cycle.**
  - **This corresponds to one micro-instruction in microprogrammed CPUs.**
  - **Examples: register operations: shift, load, clear, increment, ALU operations: add , subtract, etc.**
- **Thus: A single machine instruction may take one or more CPU cycles to complete termed as the Cycles Per Instruction (CPI).**

  Instructions Per Cycle = IPC = 1/CPI
- **Average (or effective) CPI of a program: The average CPI of all instructions executed in the program on a given CPU design.**

4th Edition: Chapter 1 (1.4, 1.7, 1.8)
3rd Edition: Chapter 4

Cycles/sec = Hertz = Hz
MHz = 10^6 Hz GHz = 10^9 Hz

Computer Performance Measures: Program Execution Time

- For a specific program compiled to run on a specific machine (CPU) "A", has the following parameters:
  - The total executed instruction count of the program. **I**
  - The average number of cycles per instruction (average CPI). **CPI**
  - Clock cycle of machine "A" **C**

  Or effective CPI
- How can one measure the performance of this machine (CPU) running this program?
  - Intuitively the machine (or CPU) is said to be faster or has better performance running this program if the total execution time is shorter.
  - Thus the inverse of the total measured program execution time is a possible performance measure or metric:

Programs/second — Seconds/program

$$\text{Performance}_A = 1 \;/\; \text{Execution Time}_A$$

How to compare performance of different machines?

What factors affect performance? How to improve performance?

Comparing Computer Performance Using Execution Time

- **To compare the performance of two machines (or CPUs) "A", "B" running a given specific program:**

$$\text{Performance}_A = 1 / \text{Execution Time}_A$$
$$\text{Performance}_B = 1 / \text{Execution Time}_B$$

- **Machine A is n times faster than machine B means (or slower? if n < 1) :**

$$\text{Speedup} = n = \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A}$$

- **Example:** **(i.e Speedup is ratio of performance, no units)**

For a given program:

Execution time on machine A: $\text{Execution}_A = 1$ **second**

Execution time on machine B: $\text{Execution}_B = 10$ **seconds**

Speedup= $\text{Performance}_A / \text{Performance}_B = \text{Execution Time}_B / \text{Execution Time}_A$

$= 10 / 1 = 10$

The performance of machine A is 10 times the performance of machine B when running this program, or: Machine A is said to be 10 times faster than machine B when running this program.

The two CPUs may target different ISAs provided the program is written in a high level language (HLL)

CPU Execution Time: The CPU Equation

- **A program is comprised of a number of instructions executed , I**
  - **Measured in: instructions/program**
- **The average instruction executed takes a number of *cycles per instruction (CPI)* to be completed.**
  - **Measured in: cycles/instruction, CPI**

Or Instructions Per Cycle (IPC):
IPC = 1/CPI

- **CPU has a fixed clock cycle time C = 1/clock rate**
  - **Measured in: seconds/cycle**

$C = 1 / f$

- **CPU execution time is the product of the above three parameters as follows:**

Executed

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

$$T = I \times CPI \times C$$

T	I	CPI	C
execution Time per program in seconds	Number of instructions executed	Average CPI for program	CPU Clock Cycle

(This equation is commonly known as the CPU performance equation)

CPU Execution Time: Example

- **A Program is running on a specific machine (CPU) with the following parameters:**
  - **Total executed instruction count: 10,000,000 instructions**
  - **Average CPI for the program: 2.5 cycles/instruction.**
  - **CPU clock rate: 200 MHz. (clock cycle = C = 5×10^{-9} seconds)** i.e 5 nanoseconds
- **What is the execution time for this program:**

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

CPU time = Instruction count x CPI x Clock cycle

= 10,000,000 x 2.5 x 1 / clock rate

= 10,000,000 x 2.5 x 5×10^{-9}

= 0.125 seconds

T = I x CPI x C

Nanosecond = nsec =ns = 10^{-9} second

Factors Affecting CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

$$T = I \times CPI \times C$$

	Instruction Count	Cycles per Instruction	Clock Cycle Time
Program	X	X	
Compiler	X	X	
Instruction Set Architecture (ISA)	X	X	
Organization (CPU Design)		X	X
Technology (VLSI)			X

Performance Comparison: Example

- From the previous example: A Program is running on a specific machine (CPU) with the following parameters:
  - Total executed instruction count, I: 10,000,000 instructions
  - Average CPI for the program: 2.5 cycles/instruction.
  - CPU clock rate: 200 MHz.

Thus: $C = 1/(200 \times 10^6) = 5 \times 10^{-9}$ seconds

- Using the same program with these changes:
  - A new compiler used: New executed instruction count, I: 9,500,000

    New CPI: 3.0
  - Faster CPU implementation: New clock rate = 300 MHz

Thus: $C = 1/(300 \times 10^6) = 3.33 \times 10^{-9}$ seconds

- What is the speedup with the changes?

$$\text{Speedup} = \frac{\text{Old Execution Time}}{\text{New Execution Time}} = \frac{I_{old} \times CPI_{old} \times \text{Clock cycle}_{old}}{I_{new} \times CPI_{new} \times \text{Clock Cycle}_{new}}$$

$$\text{Speedup} = (10{,}000{,}000 \times 2.5 \times 5 \times 10^{-9}) / (9{,}500{,}000 \times 3 \times 3.33 \times 10^{-9})$$

$$= .125 / .095 = 1.32$$

or 32 % faster after changes.

Clock Cycle = C = 1/ Clock Rate

$T = I \times CPI \times C$

Instruction Types & CPI

- **Given a program with *n* types or classes of instructions executed on a given CPU with the following characteristics:**

C_i = **Count of instructions of type$_i$ executed**

CPI_i = **Cycles per instruction for type$_i$** **i = 1, 2, …. n**

Depends on CPU Design

Then:

CPI = CPU Clock Cycles / Instruction Count I

i.e average or effective CPI

Executed

Where:

$$CPU\ clock\ cycles = \sum_{i=1}^{n} \left(CPI_i \times C_i \right)$$

Executed Instruction Count I = $\Sigma\, C_i$

T = I x CPI x C

Instruction Types & CPI: An Example

- **An instruction set has three instruction classes:**

Instruction class	CPI
A	1
B	2
C	3

For a specific CPU design

- **Two code sequences have the following instruction counts:**

	Instruction counts for instruction class		
Code Sequence	A	B	C
1	2	1	2
2	4	1	1

- **CPU cycles for sequence 1 = 2 x 1 + 1 x 2 + 2 x 3 = 10 cycles**

  CPI for sequence 1 = clock cycles / instruction count

  = 10 /5 = 2

i.e average or effective CPI

- **CPU cycles for sequence 2 = 4 x 1 + 1 x 2 + 1 x 3 = 9 cycles**

  CPI for sequence 2 = 9 / 6 = 1.5

$$CPU\ clock\ cycles = \sum_{i=1}^{n} (CPI_i \times C_i)$$

CPI = CPU Cycles / I

Instruction Frequency & CPI

- **Given a program with *n* types or classes of instructions with the following characteristics:**

i = 1, 2, …. n

C_i = Count of instructions of type$_i$ executed

CPI_i = Average cycles per instruction of type$_i$

F_i = Frequency or fraction of instruction type$_i$ executed

= C_i/ total executed instruction count = C_i/ I

Where: Executed Instruction Count I = ΣC_i

Then:

$$CPI = \sum_{i=1}^{n} \left(CPI_i \times F_i \right)$$

i.e average or effective CPI

Fraction of total execution time for instructions of type i = $\dfrac{CPI_i \times F_i}{CPI}$

T = I x CPI x C

Instruction Type Frequency & CPI: A RISC Example

Program Profile or Executed Instructions Mix

Base Machine (Reg / Reg)

Op	Freq, F_i	CPI_i	$CPI_i \times F_i$	% Time
ALU	50%	1	.5	23% = .5/2.2
Load	20%	5	1.0	45% = 1/2.2
Store	10%	3	.3	14% = .3/2.2
Branch	20%	2	.4	18% = .4/2.2
			Sum = 2.2	

Depends on CPU Design

$\frac{CPI_i \times F_i}{CPI}$

Given

Typical Mix

$$CPI = \sum_{i=1}^{n} (CPI_i \times F_i)$$

i.e average or effective CPI

CPI = .5 x 1 + .2 x 5 + .1 x 3 + .2 x 2 = 2.2

= .5 + 1 + .3 + .4

Computer Performance Measures : MIPS (Million Instructions Per Second) Rating

- **For a specific program running on a specific CPU the MIPS rating is a measure of how many millions of instructions are executed per second:**

MIPS Rating = Instruction count / (Execution Time x 10^6)

= Instruction count / (CPU clocks x Cycle time x 10^6)

= (Instruction count x Clock rate) / (Instruction count x CPI x 10^6)

= Clock rate / (CPI x 10^6)

- **Major problem with MIPS rating: As shown above the MIPS rating does not account for the count of instructions executed (I).**
  - **A higher MIPS rating in many cases may not mean higher performance or better execution time. i.e. due to compiler design variations.**
- **In addition the MIPS rating:**
  - **Does not account for the instruction set architecture (ISA) used.**
    - **Thus it cannot be used to compare computers/CPUs with different instruction sets.**
  - **Easy to abuse: Program used to get the MIPS rating is often omitted.**
    - **Often the Peak MIPS rating is provided for a given CPU which is obtained using a program comprised entirely of instructions with the lowest CPI for the given CPU design which does not represent real programs.**

T = I x CPI x C

Computer Performance Measures : MIPS (Million Instructions Per Second) Rating

- **Under what conditions can the MIPS rating be used to compare performance of different CPUs?**

- **The MIPS rating is only valid to compare the performance of different CPUs provided that the following conditions are satisfied:**

  1 **The same program is used**

  (actually this applies to all performance metrics)

  2 **The same ISA is used**

  3 **The same compiler is used**

  $\Rightarrow$ **(Thus the resulting programs used to run on the CPUs and obtain the MIPS rating are identical at the machine code level including the same instruction count)** **(binary)**

Compiler Variations, MIPS & Performance: An Example

- For a machine (CPU) with instruction classes:

Instruction class	CPI
A	1
B	2
C	3

- For a given high-level language program, two compilers produced the following executed instruction counts:

	Instruction counts (in millions) for each instruction class		
Code from:	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

- The machine is assumed to run at a clock rate of 100 MHz.

Compiler Variations, MIPS & Performance: An Example (Continued)

MIPS = Clock rate / (CPI x 10^6) = 100 MHz / (CPI x 10^6)

CPI = CPU execution cycles / Instructions count

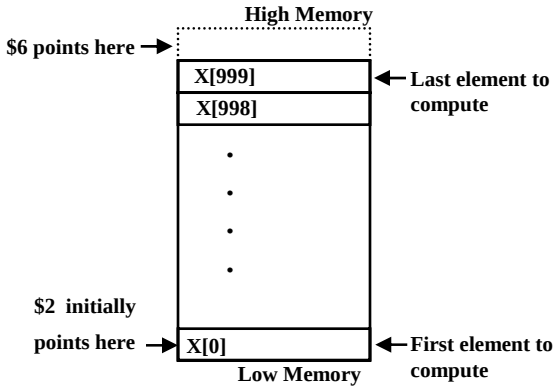
$$CPU\ clock\ cycles = \sum_{i=1}^{n} \left(CPI_i \times C_i \right)$$

CPU time = Instruction count x CPI / Clock rate

- **For compiler 1:**
  - **CPI_1 = (5 x 1 + 1 x 2 + 1 x 3) / (5 + 1 + 1) = 10 / 7 = 1.43**
  - **MIPS $Rating_1$ = 100 / (1.428 x 10^6) = 70.0 MIPS**
  - **CPU $time_1$ = ((5 + 1 + 1) x 10^6 x 1.43) / (100 x 10^6) = 0.10 seconds**
- **For compiler 2:**
  - **CPI_2 = (10 x 1 + 1 x 2 + 1 x 3) / (10 + 1 + 1) = 15 / 12 = 1.25**
  - **MIPS $Rating_2$ = 100 / (1.25 x 10^6) = 80.0 MIPS**
  - **CPU $time_2$ = ((10 + 1 + 1) x 10^6 x 1.25) / (100 x 10^6) = 0.15 seconds**

MIPS rating indicates that compiler 2 is better while in reality the code produced by compiler 1 is faster

MIPS (The ISA not the metric) Loop Performance Example



For the loop:

for (i=0; i<1000; i=i+1){

x[i] = x[i] + s; }

MIPS assembly code is given by:

```
        lw     $3,  8($1)        ; load s in $3
        addi   $6,  $2,  4000    ; $6 = address of last element + 4
loop:   lw     $4,  0($2)        ; load x[i] in $4
        add    $5,  $4, $3       ; $5 has x[i] + s
        sw     $5,  0($2)        ; store computed x[i]
        addi   $2,  $2,  4       ; increment $2 to point to next x[ ] element
        bne    $6,  $2,  loop    ; last loop iteration reached?
```

The MIPS code is executed on a specific CPU that runs at 500 MHz (clock cycle = 2ns = $2x10^{-9}$ seconds) with following instruction type CPIs :

Instruction type	CPI
ALU	4
Load	5
Store	7
Branch	3

For this MIPS code running on this CPU find:

1- Fraction of total instructions executed for each instruction type
2- Total number of CPU cycles
3- Average CPI
4- Fraction of total execution time for each instructions type
5- Execution time
6- MIPS rating , peak MIPS rating for this CPU

X[ ] array of words in memory, base address in $2 ,
s a constant word value in memory, address in $1

MIPS (The ISA) Loop Performance Example (continued)

- **The code has 2 instructions before the loop and 5 instructions in the body of the loop which iterates 1000 times,**
- **Thus: Total instructions executed, I = 5x1000 + 2 = 5002 instructions**

1 **Number of instructions executed/fraction F_i for each instruction type:**

- **ALU instructions = 1 + 2x1000 = 2001** $CPI_{ALU} = 4$ $Fraction_{ALU} = F_{ALU} = 2001/5002 = 0.4 = 40\%$
- **Load instructions = 1 + 1x1000 = 1001** $CPI_{Load} = 5$ $Fraction_{Load} = F_{Load} = 1001/5002 = 0.2 = 20\%$
- **Store instructions = 1000** $CPI_{Store} = 7$ $Fraction_{Store} = F_{Store} = 1000/5002 = 0.2 = 20\%$
- **Branch instructions = 1000** $CPI_{Branch} = 3$ $Fraction_{Branch} = F_{Branch} = 1000/5002 = 0.2 = 20\%$

2 $$CPU\ clock\ cycles = \sum_{i=1}^{n} \left(CPI_i \times C_i \right)$$

= 2001x4 + 1001x5 + 1000x7 + 1000x3 = 23009 cycles

3 **Average CPI = CPU clock cycles / I = 23009/5002 = 4.6**

Instruction type	CPI
ALU	4
Load	5
Store	7
Branch	3

4 **Fraction of execution time for each instruction type:**

- **Fraction of time for ALU instructions = CPI_{ALU} x F_{ALU} / CPI= 4x0.4/4.6 = 0.348 = 34.8%**
- **Fraction of time for load instructions = CPI_{load} x F_{load} / CPI= 5x0.2/4.6 = 0.217 = 21.7%**
- **Fraction of time for store instructions = CPI_{store} x F_{store} / CPI= 7x0.2/4.6 = 0.304 = 30.4%**
- **Fraction of time for branch instructions = CPI_{branch} x F_{branch} / CPI= 3x0.2/4.6 = 0.13 = 13%**

5 **Execution time = I x CPI x C = CPU cycles x C = 23009 x $2x10^{-9}$ =**

= $4.6x\ 10^{-5}$ seconds = 0.046 msec = 46 usec

6 **MIPS rating = Clock rate / (CPI x 10^6) = 500 / 4.6 = 108.7 MIPS**

- **The CPU achieves its peak MIPS rating when executing a program that only has instructions of the type with the lowest CPI. In this case branches with $CPI_{Branch} = 3$**
- **Peak MIPS rating = Clock rate / (CPI_{Branch} x 10^6) = 500/3 = 166.67 MIPS**

Performance Enhancement Calculations: Amdahl's Law

- The performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used
- Amdahl's Law:

Performance improvement or speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without E}}{\text{Execution Time with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}$$

– Suppose that enhancement E accelerates a fraction F of the execution time by a factor S and the remainder of the time is unaffected then:

original

$$\text{Execution Time with E} = ((1-F) + F/S) \times \text{Execution Time without E}$$

Hence speedup is given by:

$$\text{Speedup(E)} = \frac{\cancel{\text{Execution Time without E}}}{((1 - F) + F/S) \times \cancel{\text{Execution Time without E}}} = \frac{1}{(1 - F) + F/S}$$

F (Fraction of execution time enhanced) refers to original execution time before the enhancement is applied

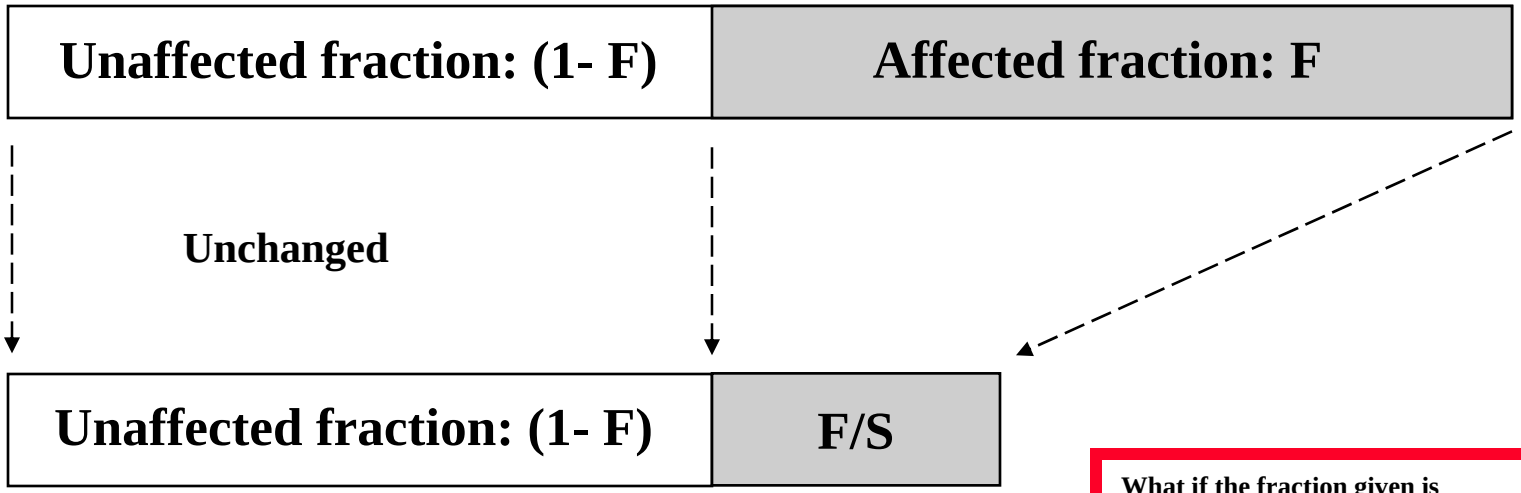
Pictorial Depiction of Amdahl's Law

Enhancement E accelerates fraction F of original execution time by a factor of S

Before:

Execution Time without enhancement E: (Before enhancement is applied)

- **shown normalized to 1 = (1-F) + F =1**



What if the fraction given is after the enhancement has been applied? How would you solve the problem? (i.e find expression for speedup)

After:

Execution Time with enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + F/S}$$

Performance Enhancement Example

- **For the RISC machine with the following instruction mix given earlier:**

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%

CPI = 2.2

- **If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:**

Fraction enhanced = F = 45% or .45

Unaffected fraction = 100% - 45% = 55% or .55

Factor of enhancement = 5/2 = 2.5

Using Amdahl's Law:

$$\text{Speedup(E)} = \frac{1}{(1 - F) + F/S} = \frac{1}{.55 + .45/2.5} = 1.37$$

An Alternative Solution Using CPU Equation

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%

CPI = 2.2

- **If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:**

Old CPI = 2.2

New CPI of load is now 2 instead of 5

New CPI = .5 x 1 + .2 x 2 + .1 x 3 + .2 x 2 = 1.6

$$\text{Speedup(E)} = \frac{\text{Original Execution Time}}{\text{New Execution Time}} = \frac{\text{Instruction count} \times \text{old CPI} \times \text{clock cycle}}{\text{Instruction count} \times \text{new CPI} \times \text{clock cycle}}$$

$$= \frac{\text{old CPI}}{\text{new CPI}} = \frac{2.2}{1.6} = 1.37$$

Which is the same speedup obtained from Amdahl's Law in the first solution.

T = I x CPI x C

Performance Enhancement Example

- **A program runs in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program four times faster?**

$$\text{Desired speedup} = 4 = \frac{100}{\text{Execution Time with enhancement}}$$

→ Execution time with enhancement = 100/4 = 25 seconds

25 seconds = (100 - 80 seconds) + 80 seconds / S

25 seconds = 20 seconds + 80 seconds / S

→ 5 = 80 seconds / S

→ S = 80/5 = 16

Alternatively, it can also be solved by finding enhanced fraction of execution time:

F = 80/100 = .8

and then solving Amdahl's speedup equation for desired enhancement factor S

$$\text{Speedup(E)} = \frac{1}{(1 - F) + F/S} = 4 = \frac{1}{(1 - .8) + .8/S} = \frac{1}{.2 + .8/s}$$

Solving for S gives S= 16

Hence multiplication should be 16 times faster to get an overall speedup of 4.

Machine = CPU

Extending Amdahl's Law To Multiple Enhancements

n enhancements each affecting a different portion of execution time

- **Suppose that enhancement E_i accelerates a fraction F_i of the original execution time by a factor S_i and the remainder of the time is unaffected then:**

i = 1, 2, …. n

$$Speedup = \frac{\text{Original Execution Time}}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right) X \text{Original Execution Time}}$$

Unaffected fraction

$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right)}$$

What if the fractions given are after the enhancements were applied? How would you solve the problem? (i.e find expression for speedup)

Note: All fractions F_i refer to original execution time before the enhancements are applied.

Amdahl's Law With Multiple Enhancements: Example

- **Three CPU performance enhancements are proposed with the following speedups and percentage of the code execution time affected:**

$Speedup_1 = S_1 = 10$	**$Percentage_1 = F_1 = 20\%$**
$Speedup_2 = S_2 = 15$	**$Percentage_1 = F_2 = 15\%$**
$Speedup_3 = S_3 = 30$	**$Percentage_1 = F_3 = 10\%$**

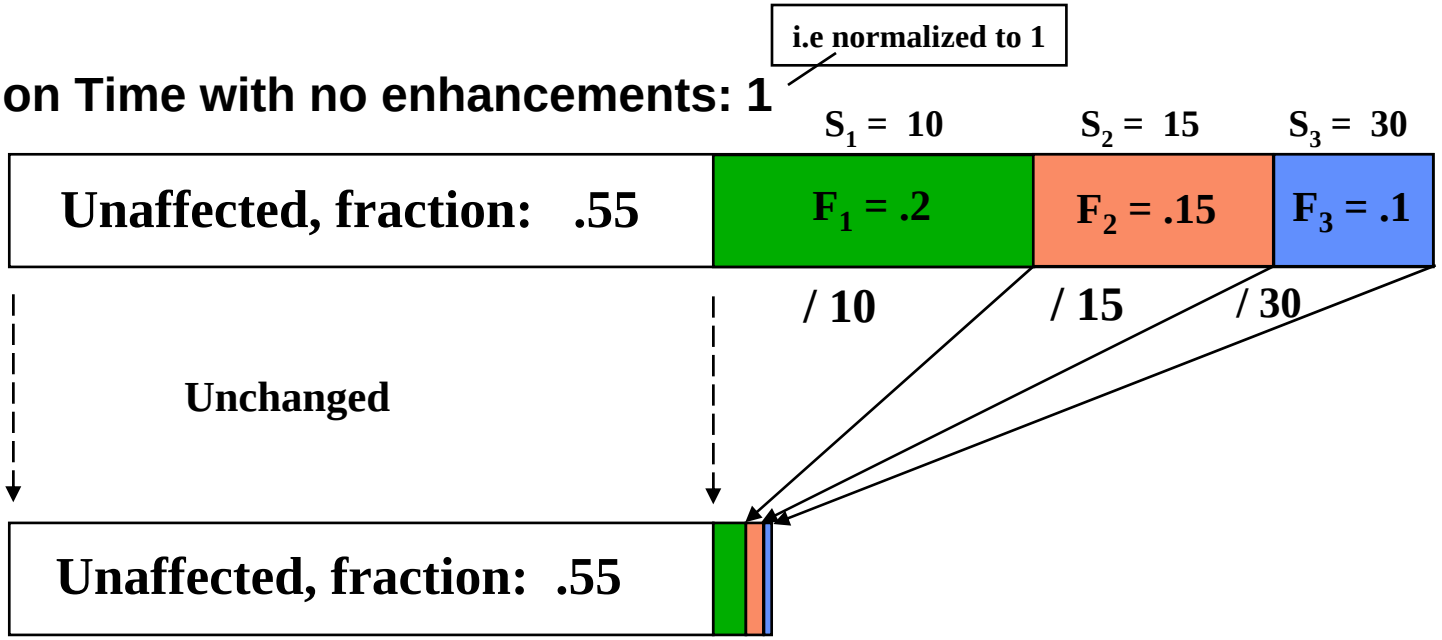
- **While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.**
- **What is the resulting overall speedup?**

$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right)}$$

- **Speedup = 1 / [(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30)]**
  = 1 / [.55 + .0333]
  = 1 / .5833 = 1.71

Pictorial Depiction of Example

Before:
Execution Time with no enhancements: 1



After:
Execution Time with enhancements: .55 + .02 + .01 + .00333 = .5833

Speedup = 1 / .5833 = 1.71

What if the fractions given are after the enhancements were applied? How would you solve the problem?

Note: All fractions refer to original execution time.

"Reverse" Multiple Enhancements Amdahl's Law

- **Multiple Enhancements Amdahl's Law assumes that the fractions given refer to original execution time.**
- **If for each enhancement S_i the fraction F_i it affects is given as a fraction of the resulting execution time after the enhancements were applied then:**

$$Speedup = \frac{\left((1 - \sum_i F_i) + \sum_i F_i \times S_i\right) X \text{Resulting ExecutionTime}}{\text{Resulting ExecutionTime}}$$

Unaffected fraction

$$Speedup = \frac{(1 - \sum_i F_i) + \sum_i F_i \times S_i}{1} = (1 - \sum_i F_i) + \sum_i F_i \times S_i$$

i.e as if resulting execution time is normalized to 1

- **For the previous example assuming fractions given refer to resulting execution time after the enhancements were applied (not the original execution time), then:**

Speedup = (1 - .2 - .15 - .1) + .2 x10 + .15 x15 + .1x30

= .55 + 2 + 2.25 + 3

= 7.8

Major CPU Design Steps

1 **Analyze instruction set to get datapath requirements:**

- **Using independent RTN, write the micro-operations required for target ISA instructions.** (1)
  - **This provides the the required datapath components and how they are connected.** (2)

2 **Select set of datapath components and establish clocking methodology (defines when storage or state elements can read and when they can be written, e.g clock edge-triggered)** + **Determine number of cycles per instruction and operations in each cycle.**

3 **Assemble datapath meeting the requirements.**

4 **Identify and define the function of all control points or signals needed by the datapath.**

- **Analyze implementation of each instruction to determine setting of control points that affects its operations.** **For each cycle of the instruction**

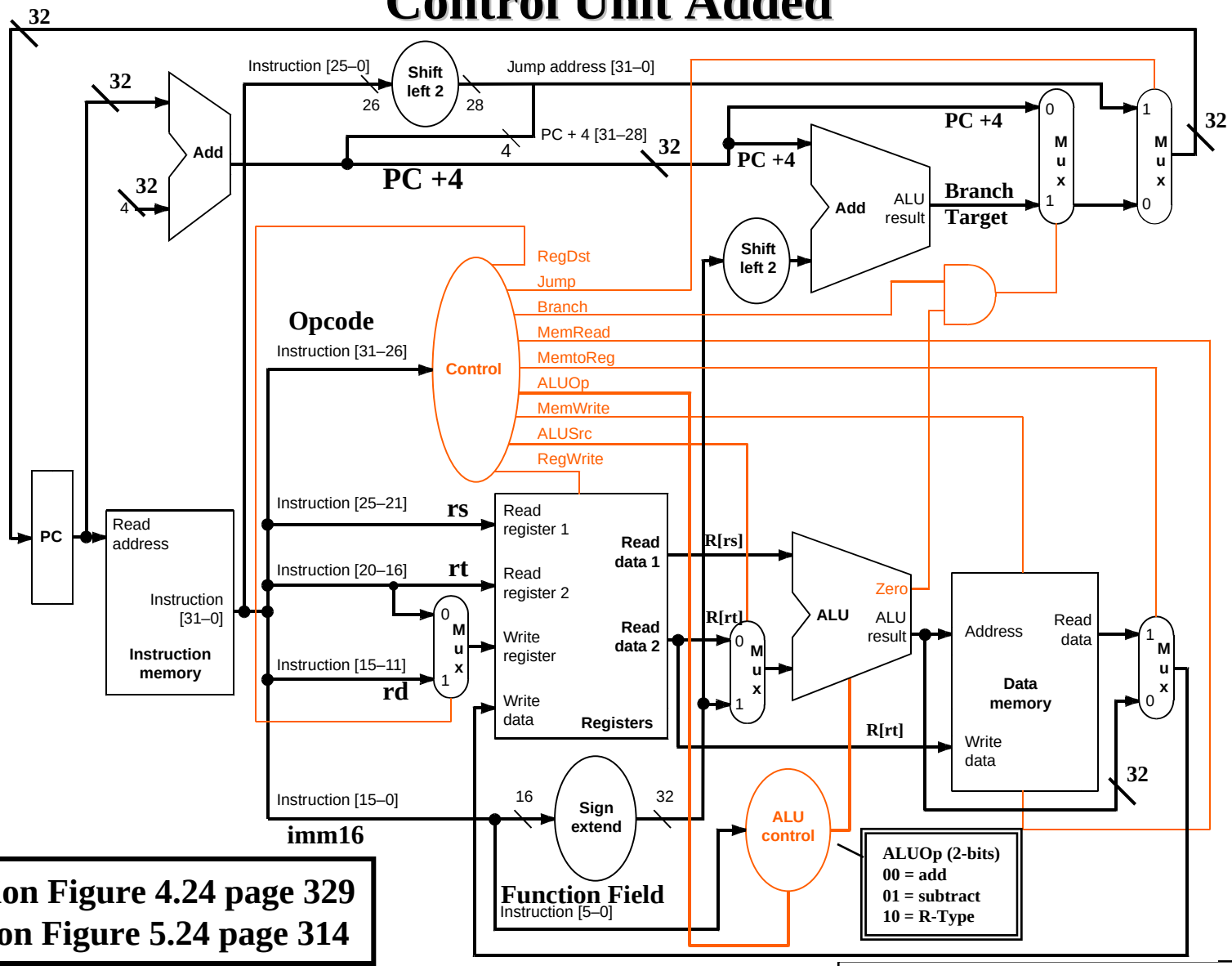
5 **Control unit design, based on micro-operation timing and control signals identified:**

- **Combinational logic: For single cycle CPU.** **e.g Any instruction completed in one cycle**
- **Hard-Wired: Finite-state machine implementation.** **i.e CPI = 1**
- **Microprogrammed.**

Datapath Design Steps

- **Write the micro-operation sequences required for a number of representative target ISA instructions using independent RTN.**
- **Independent RTN statements specify: the required datapath components and how they are connected.**
  1 2
- **From the above, create an initial datapath by determining possible destinations for each data source (i.e registers, ALU).**
  - **This establishes connectivity requirements (data paths, or connections) for datapath components.**
  - **Whenever *multiple sources* are connected to *a single input* (or destination), a multiplexor of appropriate size is added.**
    Or the size of an existing mux is increased
- **Find the worst-time propagation delay in the datapath to determine the datapath clock cycle (CPU clock cycle).**
- **Complete the micro-operation sequences for all remaining instructions adding datapath components + connections/multiplexors as needed.**

Single Cycle MIPS Datapath Extended To Handle Jump with Control Unit Added



4th Edition Figure 4.24 page 329
3rd Edition Figure 5.24 page 314

In this book version, ORI is not supported—no zero extend of immediate needed.

Control Lines Settings

(For Textbook Single Cycle Datapath including Jump)

	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0	Jump	
R-format	1	0	0	1	0	0	0	1	0	0	
lw	0	1	1	1	1	0	0	0	0	0	
sw	x	1	x	0	0	1	0	0	0	0	
beq	x	0	x	0	0	0	1	0	1	0	
J	x	x	x	0	0	0	X	x	x	1	

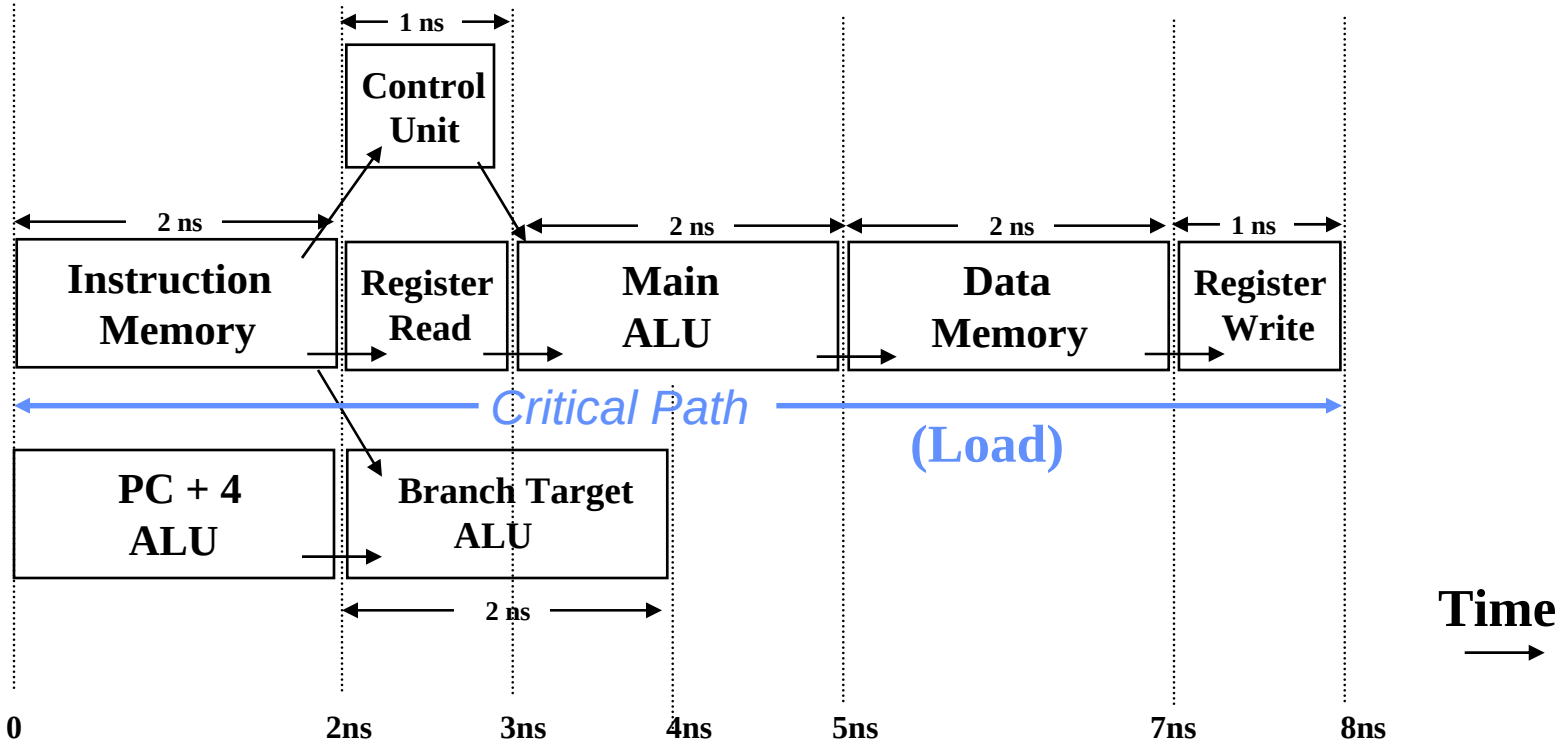
Similar to Figure 4.18 page 323 (3rd Edition Figure 5.18 page 308) with Jump instruction control line values included

Simplified Single Cycle Datapath Timing

- **Assuming the following datapath/control hardware components delays:**
  - **Memory Units: 2 ns**
  - **ALU and adders: 2 ns**
  - **Register File: 1 ns**
  - **Control Unit < 1 ns**

} **Obtained from low-level target VLSI implementation technology of components**

- **Ignoring Mux and clk-to-Q delays, critical path analysis:**



ns = nanosecond = 10^{-9} second

Performance of Single-Cycle (CPI=1) CPU

- Assuming the following datapath hardware components delays:
  - Memory Units: 2 ns
  - ALU and adders: 2 ns
  - Register File: 1 ns

Nano second, ns = 10^{-9} second

- The delays needed for each instruction type can be found :

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total Delay
ALU	2 ns	1 ns	2 ns		1 ns	6 ns
Load	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store	2 ns	1 ns	2 ns	2 ns		7 ns
Branch	2 ns	1 ns	2 ns			5 ns
Jump	2 ns					2 ns

← Load has longest delay of 8 ns thus determining the clock cycle of the CPU to be 8ns

C = 8 ns

- The clock cycle is determined by the instruction with longest delay: The load in this case which is 8 ns. Clock rate = 1 / 8 ns = 125 MHz
- A program with I = 1,000,000 instructions executed takes:

Execution Time = T = I x CPI x C = 10^6 x 1 x 8×10^{-9} = 0.008 s = 8 msec

Adding Support for jal to Single Cycle Datapath

- **The MIPS jump and link instruction, jal is used to support procedure calls by jumping to jump address (similar to j) and saving the address of the following instruction PC+4 in register $ra ($31)**

jal Address

R[31] ← PC + 4

PC ← Jump Address

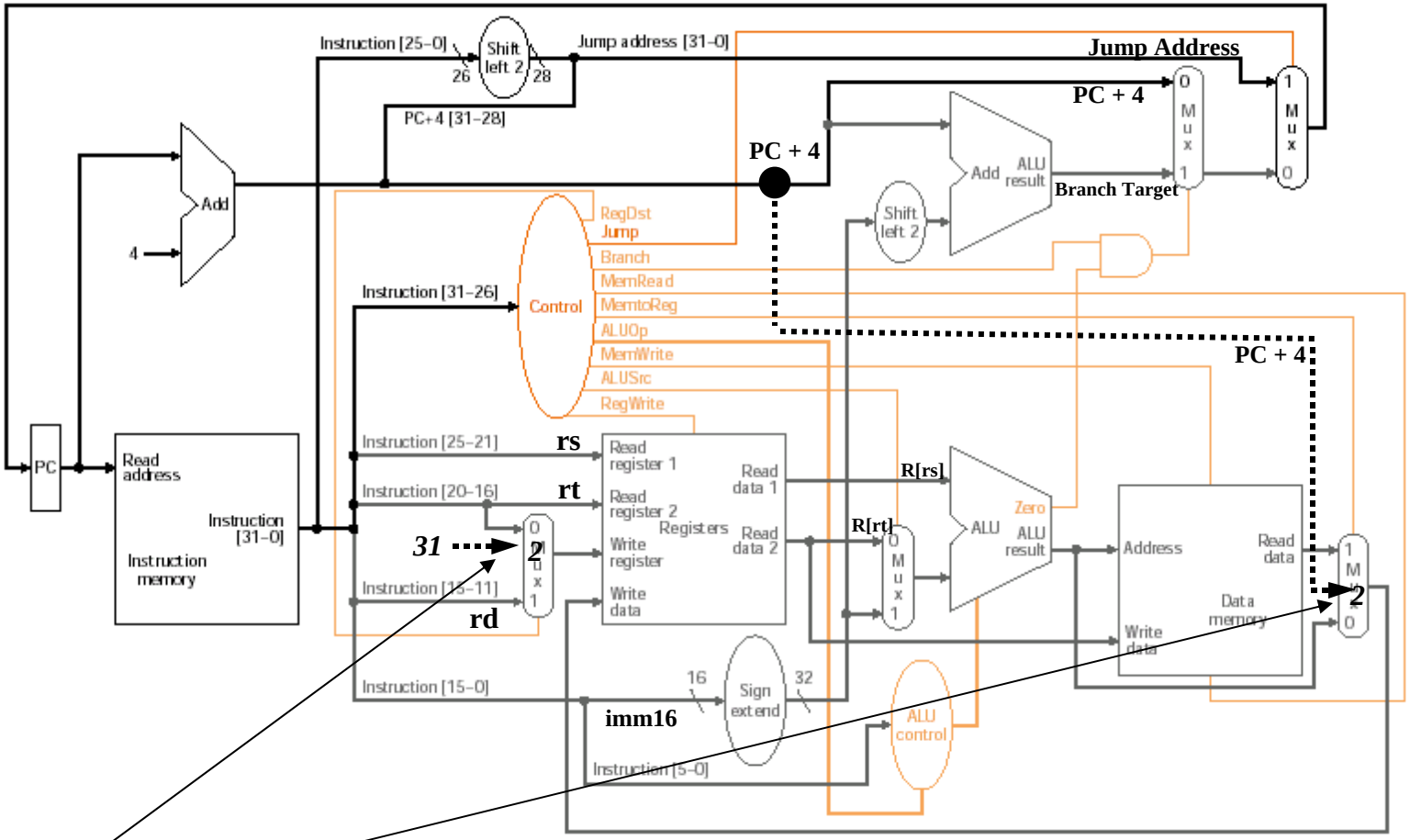
- **jal uses the j instruction format:**

op (6 bits)	Target address (26 bits)

- **We wish to add jal to the single cycle datapath in Figure 4.24 page 329 (3rd Edition Figure 5.24 page 314) . Add any necessary datapaths and control signals to the single-clock datapath and justify the need for the modifications, if any.**
- **Specify control line values for this instruction.**

jump and link, jal support to Single Cycle Datapath

Instruction Word ← Mem[PC]
R[31] ← PC + 4
PC ← Jump Address



1. **Expand the multiplexor controlled by RegDst to include the value 31 as a new input 2.**
2. **Expand the multiplexor controlled by MemtoReg to have PC+4 as new input 2.**

jump and link, jal support to Single Cycle Datapath

Adding Control Lines Settings for jal

(For Textbook Single Cycle Datapath including Jump)

RegDst Is now 2 bits

MemtoReg Is now 2 bits

	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0	Jump	
R-format	01	0	00	1	0	0	0	1	0	0	
lw	00	1	01	1	1	0	0	0	0	0	
sw	xx	1	xx	0	0	1	0	0	0	0	
beq	xx	0	xx	0	0	0	1	0	1	0	
J	xx	x	xx	0	0	0	x	x	x	1	
JAL	10	x	10	1	0	0	x	x	x	1	

R[31]

PC+ 4

PC ← Jump Address

Instruction Word ← Mem[PC]
R[31] ← PC + 4
PC ← Jump Address

Adding Support for LWR to Single Cycle Datapath

- **We wish to add a variant of lw (load word) let's call it LWR to the single cycle datapath in Figure 4.24 page 329 (3rd Edition Figure 5.24 page 314).**

LWR $rd, $rs, $rt

- **The LWR instruction is similar to lw but it sums two registers (specified by $rs, $rt) to obtain the effective load address and uses the R-Type format**

Loaded word from memory written to register rd

- **Add any necessary datapaths and control signals to the single cycle datapath and justify the need for the modifications, if any.**
- **Specify control line values for this instruction.**

Exercise 5.22: LWR (R-format LW) support to Single Cycle Datapath

Instruction Word ← Mem[PC]

PC ← PC + 4

R[rd] ← Mem[R[rs] + R[rt]]

No new components or connections are needed for the datapath just the proper control line settings

Adding Control Lines Settings for LWR

(For Textbook Single Cycle Datapath including Jump)

	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0	Jump	
R-format	1	0	0	1	0	0	0	1	0	0	
lw	0	1	1	1	1	0	0	0	0	0	
sw	x	1	x	0	0	1	0	0	0	0	
beq	x	0	x	0	0	0	1	0	1	0	
J	x	x	x	0	0	0	x	x	x	1	
LWR	1	0	1	1	1	0	0	0	0	0	

rd R[rt]

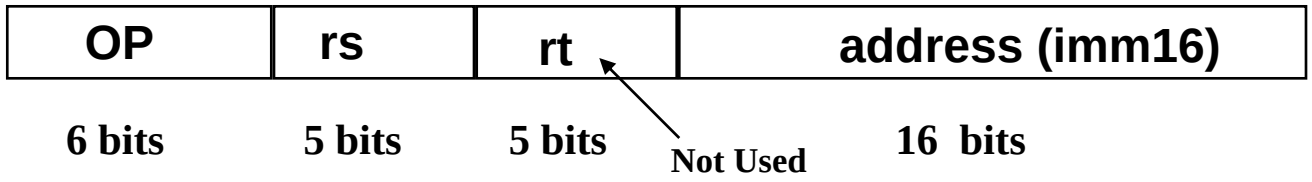
Add

Adding Support for jm to Single Cycle Datapath

- **We wish to add a new instruction jm (jump memory) to the single cycle datapath in Figure 4.24 page 329 (3rd Edition Figure 5.24 page 314).**

jm offset($rs)

- **The jm instruction loads a word from effective address (R[rs] + offset), this is similar to lw except the loaded word is put in the PC instead of register $rt.**
- **Jm used the I-format with field rt not used.**



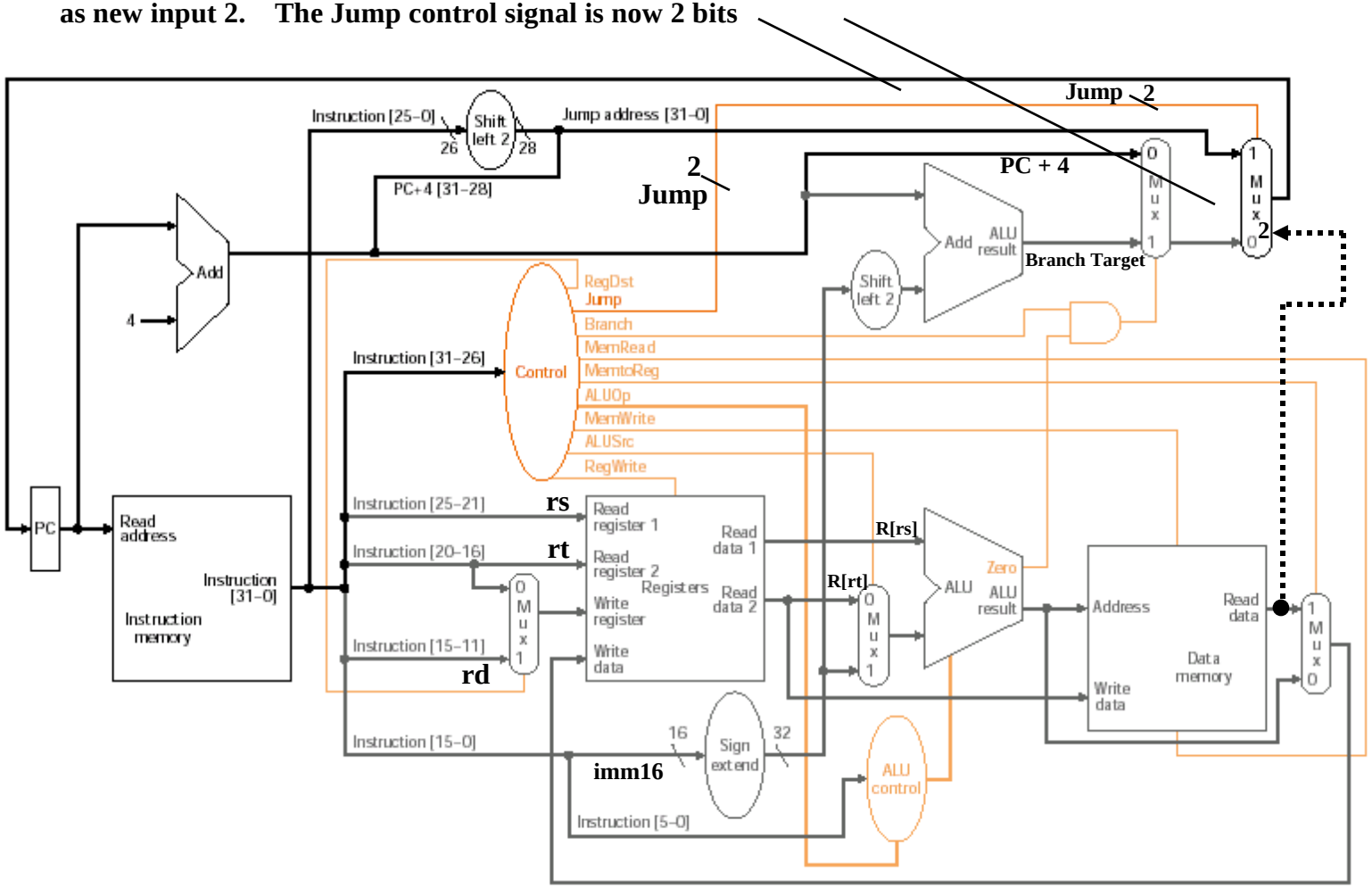
- **Add any necessary datapaths and control signals to the single cycle datapath and justify the need for the modifications, if any.**
- **Specify control line values for this instruction.**

Adding jump memory, jm support to Single Cycle Datapath

Instruction Word ← Mem[PC]

PC ← Mem[R[rs] + SignExt[imm16]]

1. **Expand the multiplexor controlled by Jump to include the Read Data (data memory output) as new input 2. The Jump control signal is now 2 bits**



Adding Control Lines Settings for jm

(For Textbook Single Cycle Datapath including Jump)

Jump
is now 2 bits

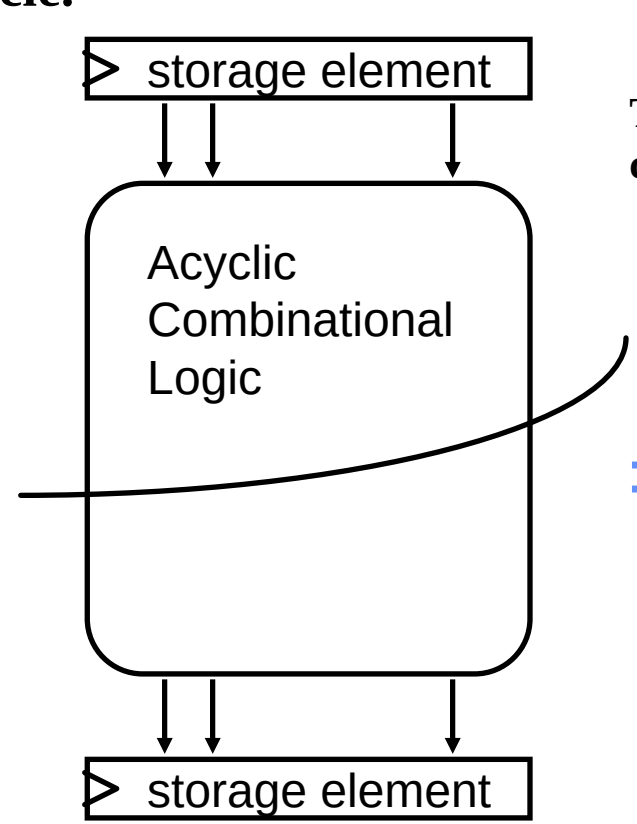
	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0	Jump	
R-format	1	0	0	1	0	0	0	1	0	00	
lw	0	1	1	1	1	0	0	0	0	00	
sw	x	1	x	0	0	1	0	0	0	00	
beq	x	0	x	0	0	0	1	0	1	00	
J	x	x	x	0	0	0	x	x	x	01	
Jm	x	1	x	0	1	0	x	0	0	10	

add

PC ← Mem[R[rs] + SignExt[imm16]]

Reducing Cycle Time: Multi-Cycle Design

- **Cut combinational dependency graph by <u>inserting registers / latches.</u>**
- **The same work is done in two or more shorter cycles, rather than one long cycle.**



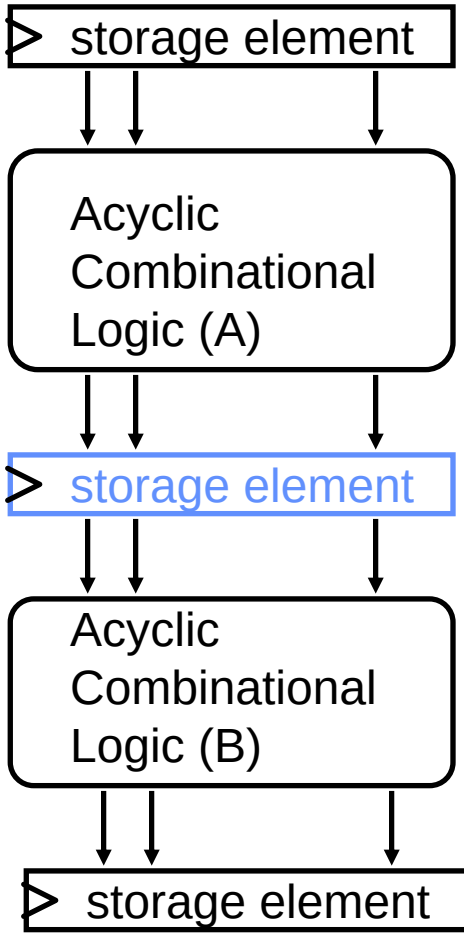
One long cycle

e.g CPI =1

Two shorter cycles

Cycle 1

=>



e.g CPI =2

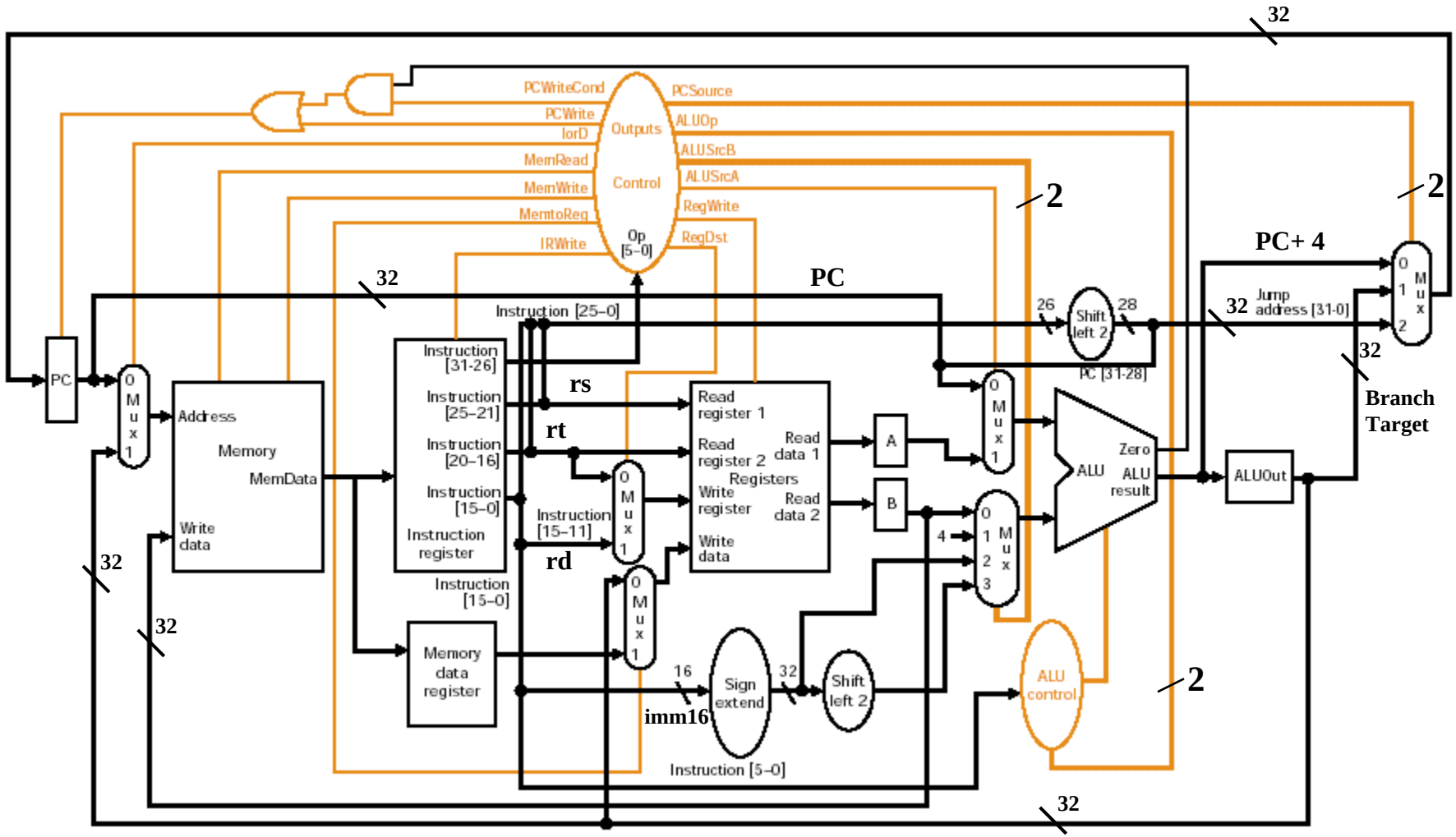
Cycle 2

Storage Element: Register or memory

Place registers to:

- **Get a balanced clock cycle length**
- **Save any results needed for the remaining cycles**

Alternative Multiple Cycle Datapath With Control Lines (3rd Edition Fig 5.28 In Textbook)



(ORI not supported, Jump supported)

3rd Edition Figure 5.28 page 323 – See handout

Operations (Dependant RTN) for Each Cycle

		R-Type	Load	Store	Branch	Jump
IF	Instruction Fetch	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4
ID	Instruction Decode	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)
EX	Execution	ALUout ← A funct B	ALUout ← A + SignEx(Imm16)	ALUout ← A + SignEx(Imm16)	Zero ← A - B Zero: PC ← ALUout	PC ← Jump Address
MEM	Memory		MDR ← Mem[ALUout]	Mem[ALUout] ← B		
WB	Write Back	R[rd] ← ALUout	R[rt] ← MDR			

Instruction Fetch (IF) & Instruction Decode (ID) cycles are common for all instructions

Multi-cycle Datapath Instruction CPI

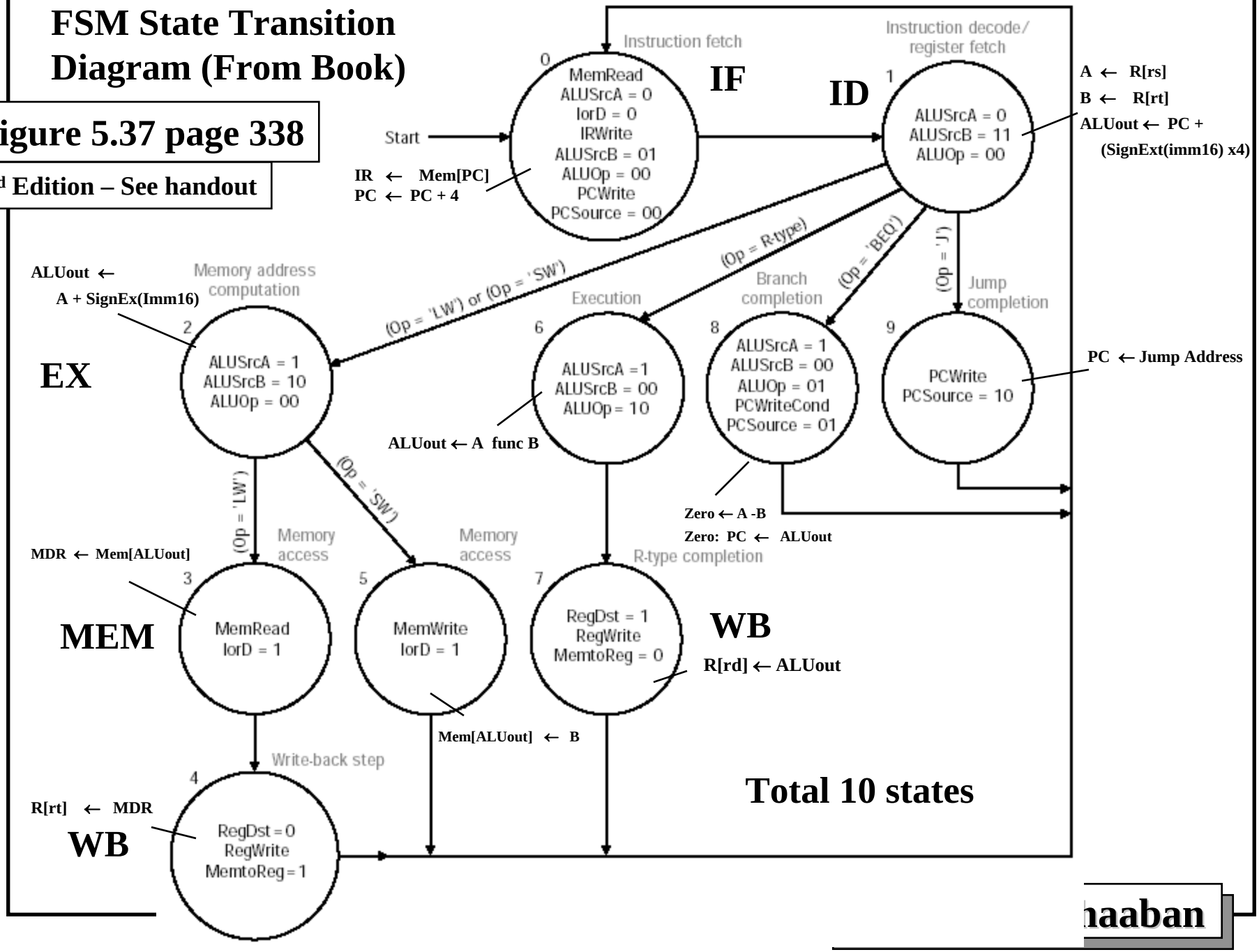
- **R-Type/Immediate: Require four cycles, CPI = 4**
  - **IF, ID, EX, WB**
- **Loads: Require five cycles, CPI = 5**
  - **IF, ID, EX, MEM, WB**
- **Stores: Require four cycles, CPI = 4**
  - **IF, ID, EX, MEM**
- **Branches/Jumps: Require three cycles, CPI = 3**
  - **IF, ID, EX**

- **Average or effective program CPI: $3 \leq CPI \leq 5$ depending on program profile (instruction mix).**

FSM State Transition Diagram (From Book)

Figure 5.37 page 338

3rd Edition – See handout



MIPS Multi-cycle Datapath Performance Evaluation

- **What is the average CPI?**
  - **State diagram gives CPI for each instruction type**
  - **Workload (program) below gives frequency of each type**

Type	CPI_i for type	Frequency	CPI_i x $freqI_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
		Average CPI:	4.1

Better than CPI = 5 if all instructions took the same number of clock cycles (5).

Adding Support for swap to Multi Cycle Datapath

- **You are to add support for a new instruction, swap that exchanges the values of two registers to the MIPS multicycle datapath of Figure 5.28 on page 232**

  swap $rs, $rt

  R[rt] ← R[rs]
  R[rs] ← R[rt]

- **Swap used the R-Type format with:**

  the value of field rs = the value of field rd

- **Add any necessary datapaths and control signals to the multicycle datapath. Find a solution that minimizes the number of clock cycles required for the new instruction <u>without modifying the register file</u>. Justify the need for the modifications, if any.**

  i.e No additional register write ports

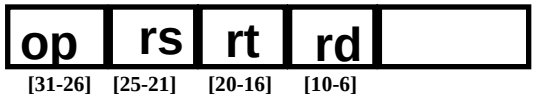
- **Show the necessary modifications to the multicycle control finite state machine of Figure 5.38 on page 339 when adding the swap instruction. For each new state added, provide the dependent RTN and active control signal values.**

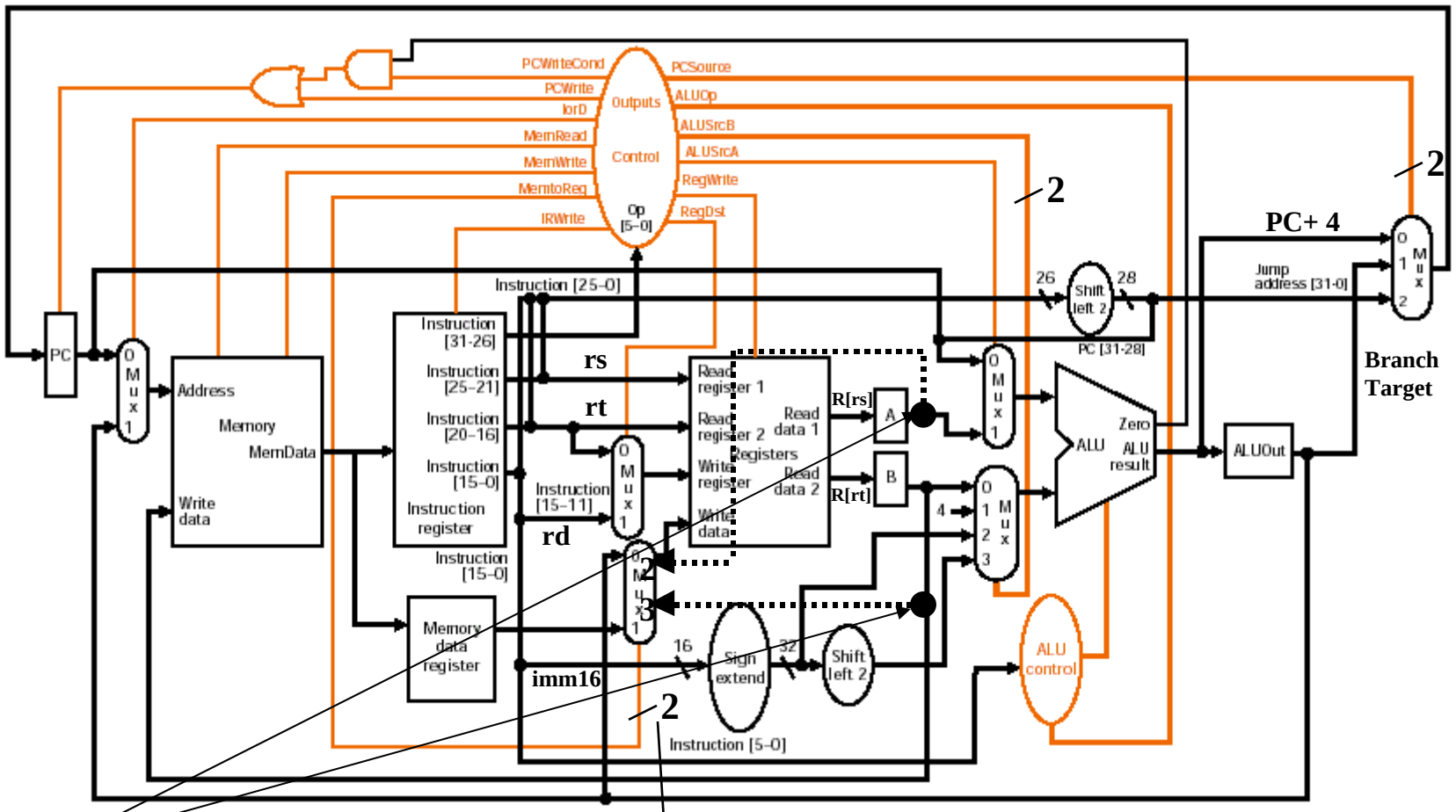
Adding swap Instruction Support to Multi Cycle Datapath

Swap $rs, $rt **R[rt] ← R[rs]**

R[rs] ← R[rt]

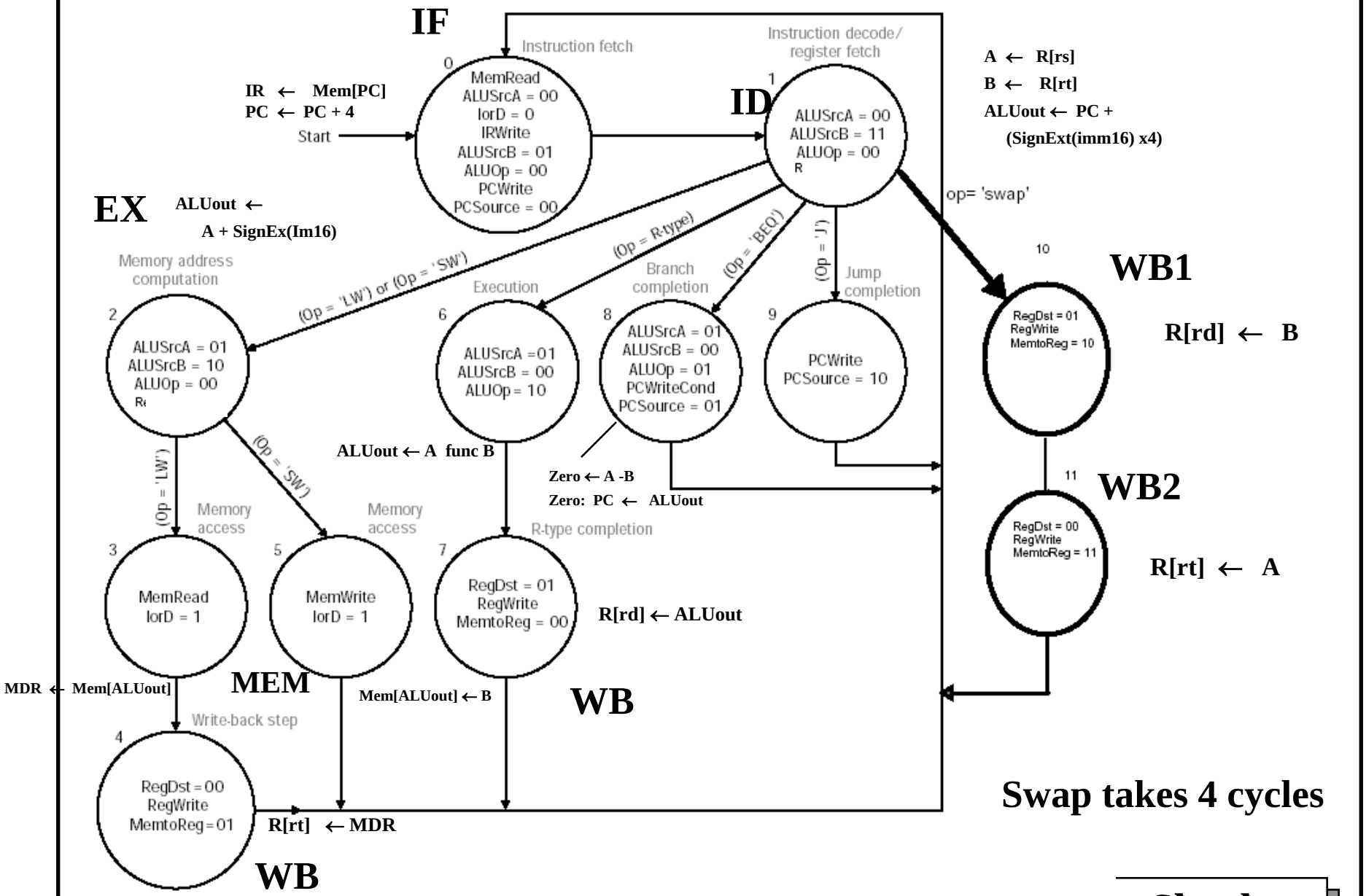
We assume here rs = rd in instruction encoding





The outputs of A and B should be connected to the multiplexor controlled by MemtoReg if one of the two fields (rs and rd) contains the name of one of the registers being swapped. The other register is specified by rt. The MemtoReg control signal becomes two bits.

Adding swap Instruction Support to Multi Cycle Datapath



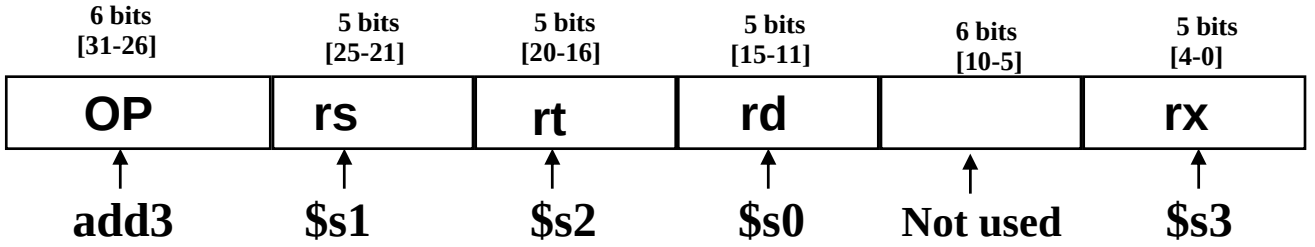
Adding Support for add3 to Multi Cycle Datapath

- **You are to add support for a new instruction, add3, that adds the values of three registers, to the MIPS multicycle datapath of Figure 5.28 on page 232**
  For example:

  add3 $s0,$s1, $s2, $s3

  Register $s0 gets the sum of $s1, $s2 and $s3.

  The instruction encoding uses a modified R-format, with an additional register specifier rx added replacing the five low bits of the "funct" field.



- **Add necessary datapath components, connections, and control signals to the multicycle datapath <u>without modifying the register bank or adding additional ALUs</u>. Find a solution that minimizes the number of clock cycles required for the new instruction. Justify the need for the modifications, if any.**
- **Show the necessary modifications to the multicycle control finite state machine of Figure 5.38 on page 339 when adding the add3 instruction. For each new state added, provide the dependent RTN and active control signal values.**

add3 instruction support to Multi Cycle Datapath

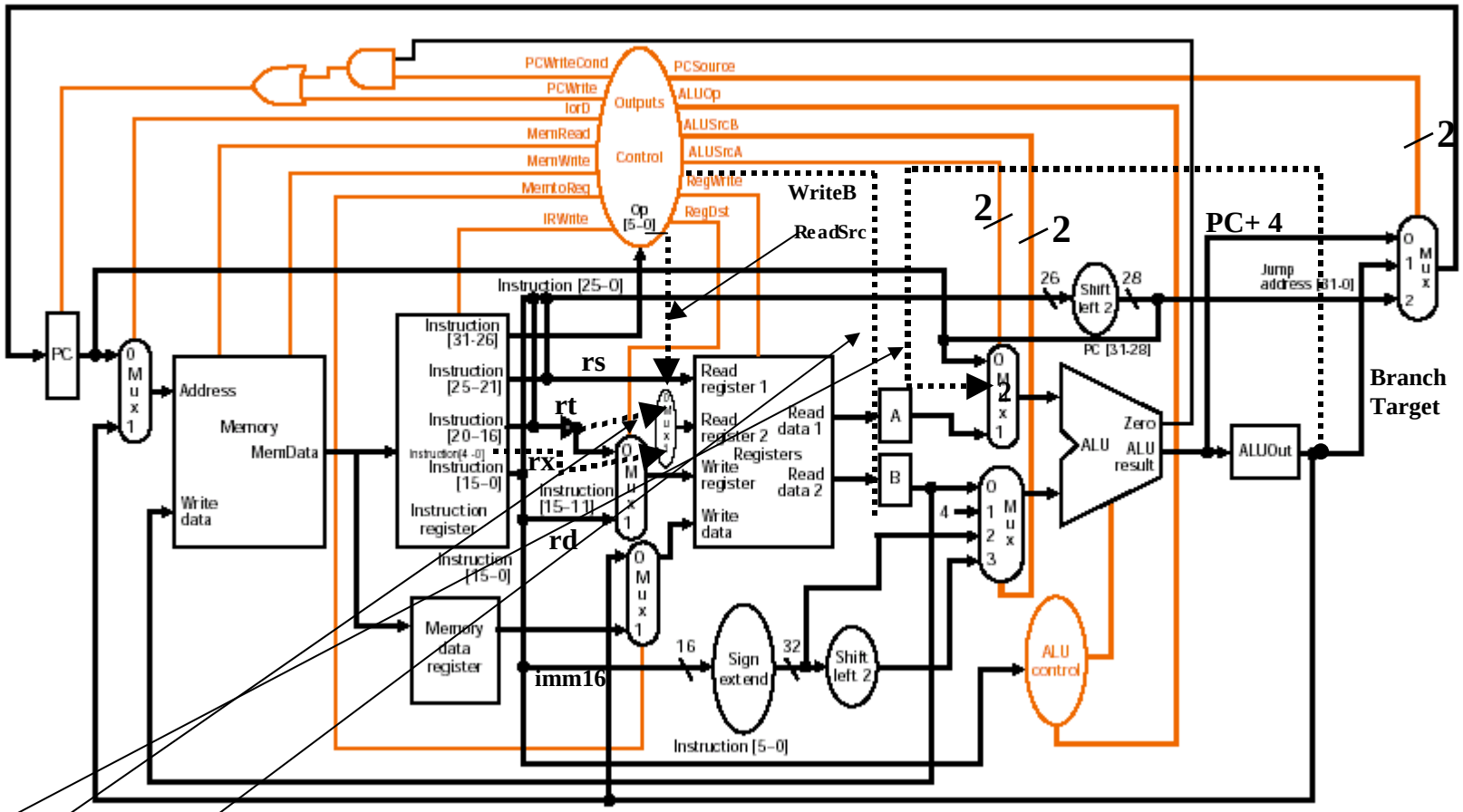
Add3 $rd, $rs, $rt, $rx

R[rd] ← R[rs] + R[rt] + R[rx]

rx is a new register specifier in field [0-4] of the instruction
No additional register read ports or ALUs allowed

Modified R-Format

op	rs	rt	rd		rx
[31-26]	[25-21]	[20-16]	[10-6]		[4-0]



1. ALUout is added as an extra input to first ALU operand MUX to use the previous ALU result as an input for the second addition.
2. A multiplexor should be added to select between rt and the new field rx containing register number of the 3rd operand (bits 4-0 for the instruction) for input for Read Register 2.
This multiplexor will be controlled by a new one bit control signal called ReadSrc.
3. WriteB control line added to enable writing R[rx] to B

add3 instruction support to Multi Cycle Datapath