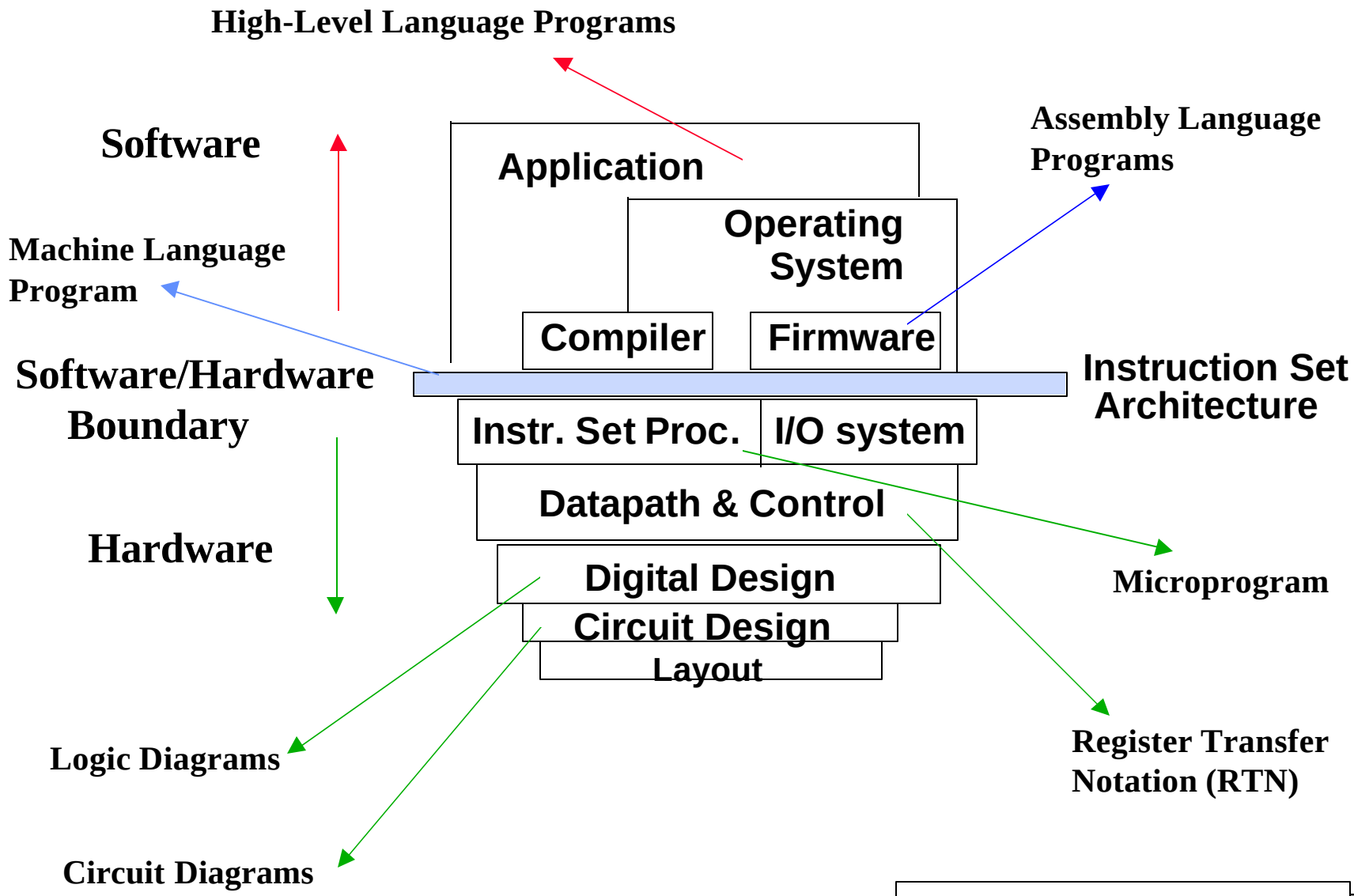
CPU Organization

- **Datapath Design:**
  - **Capabilities & performance characteristics of principal Functional Units (FUs):**
  - **(e.g., Registers, ALU, Shifters, Logic Units, ...)**
  - **Ways in which these components are interconnected (buses connections, multiplexors, etc.).**
  - **How information flows between components.**
- **Control Unit Design:**
  - **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).**

Hierarchy of Computer Architecture



Instruction Set Architecture (ISA)

"... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation."
– Amdahl, Blaaw, and Brooks, 1964.

The instruction set architecture is concerned with:

- **Organization of programmable storage (memory & registers): Includes the amount of addressable memory and number of available registers.**
- **Data Types & Data Structures: Encodings & representations.**
- **Instruction Set: What operations are specified.**
- **Instruction formats and encoding.**
- **Modes of addressing and accessing data items and instructions**
- **Exceptional conditions.**

Types of Instruction Set Architectures
According To Operand Addressing Fields

Memory-To-Memory Machines:

- **Operands obtained from memory and results stored back in memory by any instruction that requires operands.**
- **No local CPU registers are used in the CPU datapath.**
- **Include:**
  - **The 4 Address Machine.**
  - **The 3-address Machine.**
  - **The 2-address Machine.**

The 1-address (Accumulator) Machine:

- **A single local CPU special-purpose register (accumulator) is used as the source of one operand and as the result destination.**

The 0-address or Stack Machine:

- **A push-down stack is used in the CPU.**

General Purpose Register (GPR) Machines:

- **The CPU datapath contains several local general-purpose registers which can be used as operand sources and as result destinations.**
- **A large number of possible addressing modes.**
- **Load-Store or Register-To-Register Machines: GPR machines where only data movement instructions (loads, stores) can obtain operands from memory and store results to memory.**

Expression Evaluation Example with 3-, 2-, 1-, 0-Address, And GPR Machines

For the expression A = (B + C) * D - E where A-E are in memory

3-Address	2-Address	1-Address Accumulator	0-Address Stack	GPR	
				Register-Memory	Load-Store

add A, B, C
mul A, A, D
sub A, A, E
	load A, B
	add A, C
	mul A, D
	sub A, E
		load B
		add C
		mul D
		sub E
		store A
			push B
			push C
			add
			push D
			mul
			push E
			sub
			pop A
				load R1, B
				add R1, C
				mul R1, D
				sub R1, E
				store A, R1
					load R1, B
					load R2, C
					add R3, R1, R2
					load R1, D
					mul R3, R3, R1
					load R1, E
					sub R3, R3, R1
					store A, R3

3 instructions Code size: 30 bytes 9 memory accesses	4 instructions Code size: 28 bytes 12 memory accesses	5 instructions Code size: 20 bytes 5 memory accesses	8 instructions Code size: 23 bytes 5 memory accesses	5 instructions Code size: about 22 bytes 5 memory accesses	8 instructions Code size: about 29 bytes 5 memory accesses

Typical ISA Addressing Modes

Addressing Mode	Sample Instruction	Meaning
Register	Add R4, R3	R4 ← R4 + R3
Immediate	Add R4, #3	R4 ← R4 + 3
Displacement	Add R4, 10 (R1)	R4 ← R4 + Mem[10+ R1]
Indirect	Add R4, (R1)	R4 ← R4 + Mem[R1]
Indexed	Add R3, (R1 + R2)	R3 ← R3 +Mem[R1 + R2]
Absolute	Add R1, (1001)	R1 ← R1 + Mem[1001]
Memory indirect	Add R1, @ (R3)	R1 ← R1 + Mem[Mem[R3]]
Autoincrement	Add R1, (R2) +	R1 ← R1 + Mem[R2]
		R2 ← R2 + d
Autodecrement	Add R1, - (R2)	R2 ← R2 - d
		R1 ← R1 + Mem[R2]
Scaled	Add R1, 100 (R2) [R3]	R1 ← R1+ Mem[100+ R2 + R3*d]

Complex Instruction Set Computer (CISC)

- **Emphasizes doing more with each instruction.**
- **Motivated by the high cost of memory and hard disk capacity when original CISC architectures were proposed:**
  - **When M6800 was introduced: 16K RAM = $500, 40M hard disk = $ 55, 000**
  - **When MC68000 was introduced: 64K RAM = $200, 10M HD = $5,000**
- **Original CISC architectures evolved with faster, more complex CPU designs, but backward instruction set compatibility had to be maintained.**
- **Wide variety of addressing modes:**
    - **14 in MC68000, 25 in MC68020**
- **A number instruction modes for the location and number of operands:**
    - **The VAX has 0- through 3-address instructions.**
- **Variable-length or hybrid instruction encoding is used.**

Reduced Instruction Set Computer (RISC)

- Focuses on reducing the number and complexity of instructions of the machine.
- Reduced number of cycles needed per instruction.
  - Goal: At least one instruction completed per clock cycle.
- Designed with CPU instruction pipelining in mind.
- Fixed-length instruction encoding.
- Only load and store instructions access memory.
- Simplified addressing modes.
  - Usually limited to immediate, register indirect, register displacement, indexed.
- Delayed loads and branches.
- Prefetch and speculative execution.
- Examples: MIPS, HP-PA, UltraSpark, Alpha, PowerPC.

RISC ISA Example:

MIPS R3000

Instruction Categories:

- Load/Store.
- Computational.
- Jump and Branch.
- Floating Point (using coprocessor).
- Memory Management.
- Special.

4 Addressing Modes:

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

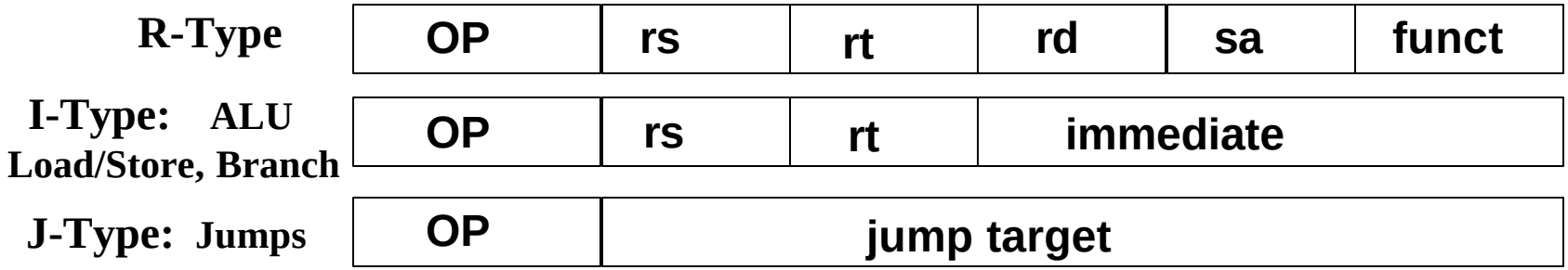
Operand Sizes:

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

Registers

R0 - R31
PC
HI
LO

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



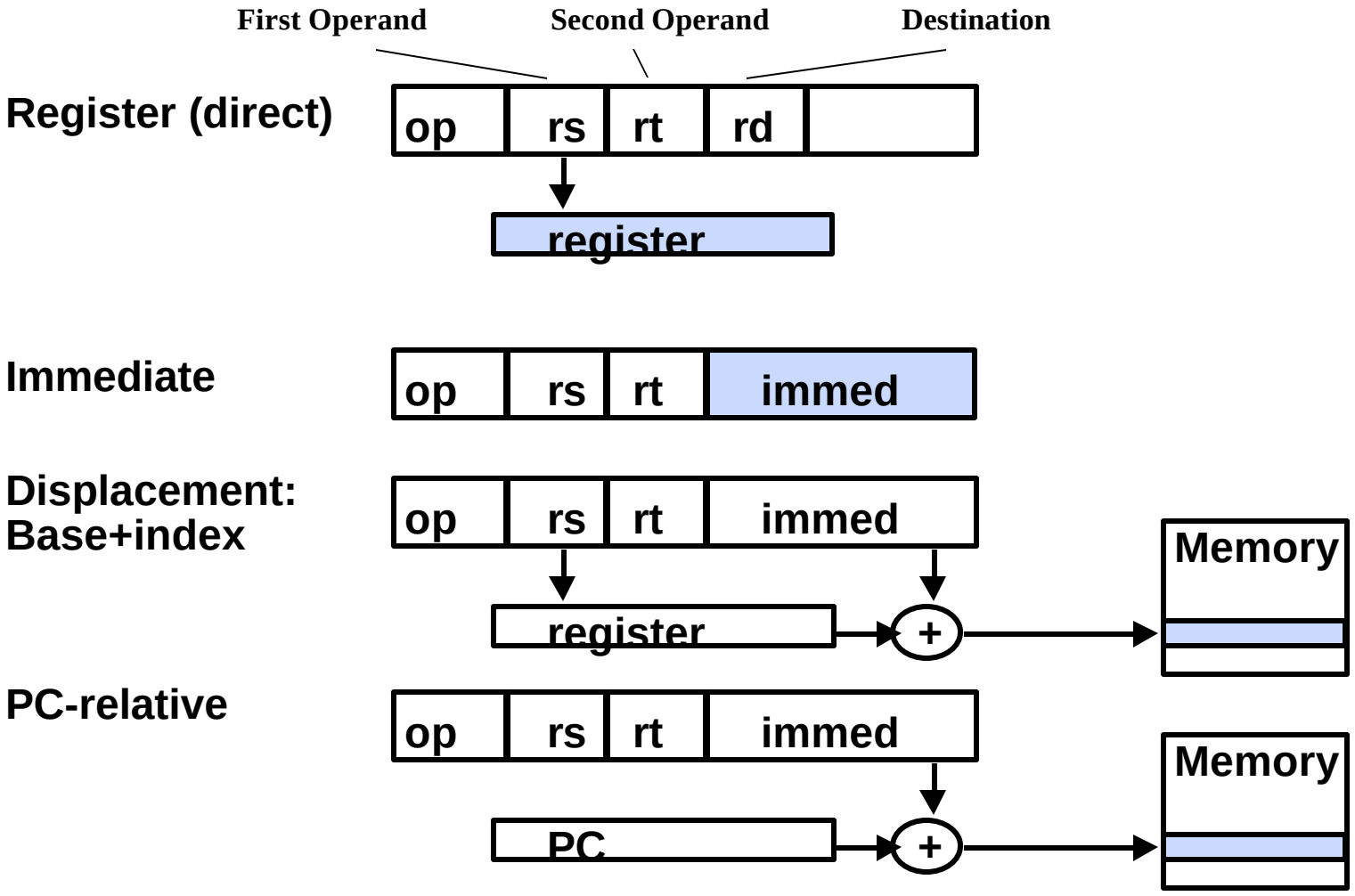
MIPS Register Usage/Naming Conventions

- **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 Addressing Modes/Instruction Formats

- All instructions 32 bits wide



MIPS Arithmetic Instructions Examples

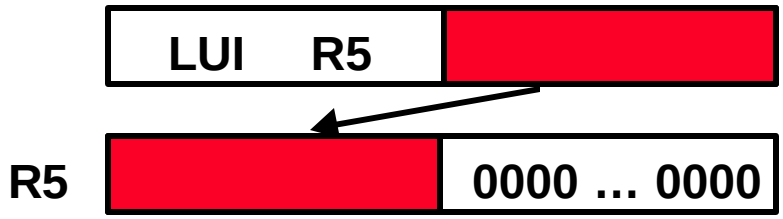
Instruction	***Example***	***Meaning***	***Comments***
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; 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 Arithmetic Instructions Examples

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 data transfer instructions Examples

Instruction	***Comment***
sw 500($4), $3	Store word
sh 502($2), $3	Store half
sb 41($3), $2	Store byte
lw $1, 30($2)	Load word
lh $1, 40($3)	Load halfword
lhu $1, 40($3)	Load halfword 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)



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: C Assignment With Variable Index To MIPS

- **For the C statement with a variable array index:**

```
g = h + A[i];
```

- **Assume: g: $s1, h: $s2, i: $s4, base address of A[ ]: $s3**
- **Steps:**
  - **Turn index i to a byte offset by multiplying by four or by addition as done here: i + i = 2i, 2i + 2i = 4i**
  - **Next add 4i to base address of A**
  - **Load A[i] into a temporary register.**
  - **Finally add to h and put sum in g**
- **MIPS Instructions:**

```
add $t1,$s4,$s4     # $t1 = 2*i
add $t1,$t1,$t1     # $t1 = 4*i
add $t1,$t1,$s3     #$t1 = address of A[i]
lw $t0,0($t1)       # $t0 = A[i]
add $s1,$s2,$t0     # g = h + A[i]
```

Example: While C Loop to MIPS

- **While loop in C:**

```
while (save[i]==k)
        i = i + j;
```

- **Assume MIPS register mapping:**

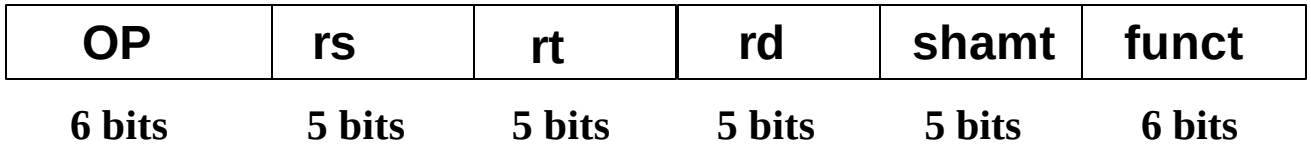
i: $s3, j: $s4, k: $s5, base of save[ ]: $s6

- **MIPS Instructions:**

```
Loop:  add $t1,$s3,$s3     # $t1 = 2*i
       add $t1,$t1,$t1     # $t1 = 4*i
       add $t1,$t1,$s6     # $t1 = Address
       lw  $t1,0($t1)      # $t1 = save[i]
       bne $t1,$s5,Exit    # goto Exit
                           # if save[i]!=k
       add $s3,$s3,$s4     # i = i + j
       j   Loop            # goto Loop
Exit:
```

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.**

Examples:

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

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

MIPS ALU I-Type Instruction Fields

I-Type ALU instructions that use two registers and an immediate value
Loads/stores, conditional branches.

6 bits **5 bits** **5 bits** **16 bits**

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

Source operand register in rs

Result register in rt

Examples:

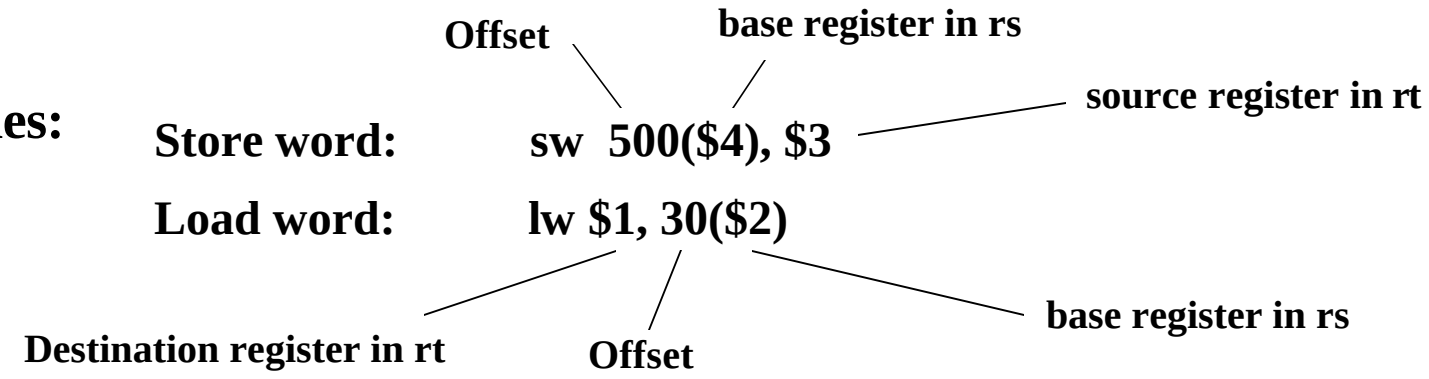
add immediate:	**addi $1,$2,100**	**Constant operand in immediate**
and immediate	**andi $1,$2,10**	

MIPS Load/Store I-Type Instruction Fields

6 bits 5 bits 5 bits 16 bits

- op: Opcode, operation of the instruction.
  - For load op = 35, for store 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:



MIPS Branch I-Type Instruction Fields

6 bits 5 bits 5 bits 16 bits

- 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.

Register in rt

Register in rs

offset in bytes equal to instruction field address x 4

Examples:

Branch on equal	beq $1,$2,100
Branch on not equal	bne $1,$2,100

MIPS J-Type Instruction Fields

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

- **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:

Branch on equal **j 10000**

Branch on not equal **jal 10000**

PC(31-28)

Computer Performance Evaluation: Cycles Per Instruction (CPI)

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

  where: Clock rate = 1 / clock cycle

- **A computer machine instruction is comprised of a number of elementary or micro operations which vary in number and complexity depending on the instruction and the exact CPU organization and implementation.**
  - **A micro operation is an elementary hardware operation that can be performed during one 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 cycles to complete termed as the Cycles Per Instruction (CPI).**

Computer Performance Measures: Program Execution Time

- For a specific program compiled to run on a specific machine "A", the following parameters are provided:
  - The total instruction count of the program.
  - The average number of cycles per instruction (average CPI).
  - Clock cycle of machine "A"
- How can one measure the performance of this machine running this program?
  - Intuitively the machine 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:

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

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 "A", "B" running a given 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:

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

- Example:

For a given program:

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

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

$$\text{Performance}_A / \text{Performance}_B = \text{Execution Time}_B / \text{Execution Time}_A = 10 / 1 = 10$$

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.

CPU Execution Time: The CPU Equation

- **A program is comprised of a number of instructions, I**
  - **Measured in: instructions/program**
- **The average instruction takes a number of cycles per instruction (CPI) to be completed.**
  - **Measured in: cycles/instruction, CPI**
- **CPU has a fixed clock cycle time C = 1/clock rate**
  - **Measured in: seconds/cycle**
- **CPU execution time is the product of the above three parameters as follows:**

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

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

CPU Execution Time: Example

- **A Program is running on a specific machine with the following parameters:**
  - **Total instruction count: 10,000,000 instructions**
  - **Average CPI for the program: 2.5 cycles/instruction.**
  - **CPU clock rate: 200 MHz.**
- **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}

= .125 seconds

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		X	X
Technology			X

Performance Comparison: Example

- From the previous example: A Program is running on a specific machine with the following parameters:
  - Total instruction count: 10,000,000 instructions
  - Average CPI for the program: 2.5 cycles/instruction.
  - CPU clock rate: 200 MHz.
- Using the same program with these changes:
  - A new compiler used: New instruction count 9,500,000

    New CPI: 3.0
  - Faster CPU implementation: New clock rate = 300 MHZ
- What is the speedup with the changes?

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

$$\text{Speedup} = (10{,}000{,}000 \times 2.5 \times 5\times10^{-9}) / (9{,}500{,}000 \times 3 \times 3.33\times10^{-9})$$

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

or 32 % faster after changes.

Instruction Types & CPI

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

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

CPI_i = **Cycles per instruction for type$_i$**

Then:

CPI = CPU Clock Cycles / Instruction Count I

Where:

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

$$\text{Instruction Count } I = \Sigma C_i$$

Instruction Types & CPI: An Example

- An instruction set has three instruction classes:

Instruction class	CPI
A	1
B	2
C	3

- 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

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

  CPI for sequence 2 = 9 / 6 = 1.5

Instruction Frequency & CPI

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

  C_i = Count of instructions of type$_i$

  CPI_i = Average cycles per instruction of type$_i$

  F_i = Frequency or fraction of instruction type$_i$

  = C_i/ total instruction count

Then:

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

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

Instruction Type Frequency & CPI: A RISC Example

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

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

Typical Mix

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

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

Computer Performance Measures : MIPS (Million Instructions Per Second)

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

$$\text{MIPS} = \text{Instruction count} / (\text{Execution Time} \times 10^6)$$
$$= \text{Instruction count} / (\text{CPU clocks} \times \text{Cycle time} \times 10^6)$$
$$= (\text{Instruction count} \times \text{Clock rate}) / (\text{Instruction count} \times \text{CPI} \times 10^6)$$
$$= \text{Clock rate} / (\text{CPI} \times 10^6)$$

- **Faster execution time usually means faster MIPS rating.**
- **Problems with MIPS rating:**
  - **No account for the instruction set used.**
  - **Program-dependent: A single machine does not have a single MIPS rating since the MIPS rating may depend on the program used.**
  - **Easy to abuse: Program used to get the MIPS rating is often omitted.**
  - **Cannot be used to compare computers with different instruction sets.**
  - **A higher MIPS rating in some cases may not mean higher performance or better execution time. i.e. due to compiler design variations.**

Compiler Variations, MIPS & Performance: An Example

- For a machine with instruction classes:

Instruction class	CPI
A	1
B	2
C	3

- For a given program, two compilers produced the following 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**
  - **MIP_1 = 100 / (1.428 x 10^6) = 70.0**
  - **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**
  - **MIP_2 = 100 / (1.25 x 10^6) = 80.0**
  - **CPU $time_2$ = ((10 + 1 + 1) x 10^6 x 1.25) / (100 x 10^6) = 0.15 seconds**

Computer Performance Measures :
MFOLPS (Million FLOating-Point Operations Per Second)

- **A floating-point operation is an addition, subtraction, multiplication, or division operation applied to numbers represented by a single or a double precision floating-point representation.**
- **MFLOPS, for a specific program running on a specific computer, is a measure of millions of floating point-operation (megaflops) per second:**

$$\text{MFLOPS} = \text{Number of floating-point operations} \;/\; (\text{Execution time} \times 10^6)$$

- **MFLOPS is a better comparison measure between different machines than MIPS.**
- **Program-dependent: Different programs have different percentages of floating-point operations present. i.e compilers have no floating-point operations and yield a MFLOPS rating of zero.**
- **Dependent on the type of floating-point operations present in the program.**

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:

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

Hence speedup is given by:

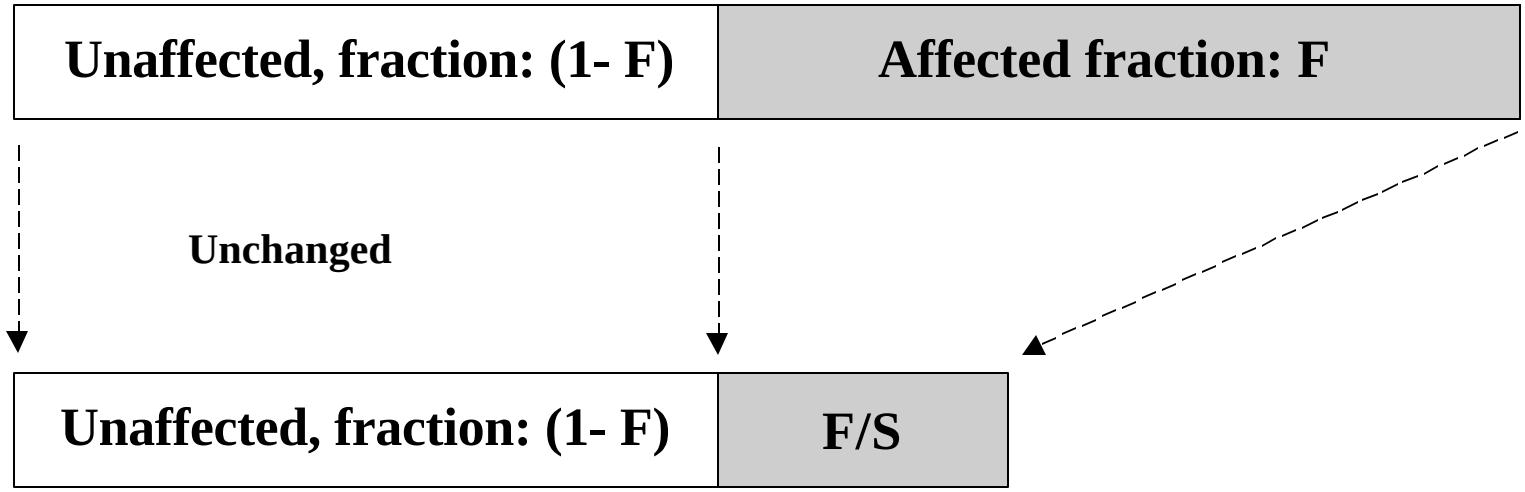
$$\text{Speedup(E)} = \frac{\cancel{\text{Execution Time without E}}}{((1 - F) + F/S) \times \cancel{\text{Execution Time without E}}} = \frac{1}{(1 - F) + F/S}$$

Note: All fractions here refer to original execution time.

Pictorial Depiction of Amdahl's Law

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

Before:
Execution Time without enhancement E:



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 = .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.

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?**

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

$\rightarrow$ **Execution time with enhancement = 25 seconds**

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

25 seconds = 20 seconds + 80 seconds / n

$\rightarrow$ **5 = 80 seconds / n**

$\rightarrow$ **n = 80/5 = 16**

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

Extending Amdahl's Law To Multiple Enhancements

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

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

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

Note: All fractions refer to original execution time.

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:**

  $Speedup_1 = S_1 = 10$ **$Percentage_1 = F_1 = 20\%$**

  $Speedup_2 = S_2 = 15$ **$Percentage_1 = F_2 = 15\%$**

  $Speedup_3 = S_3 = 30$ **$Percentage_1 = F_3 = 10\%$**

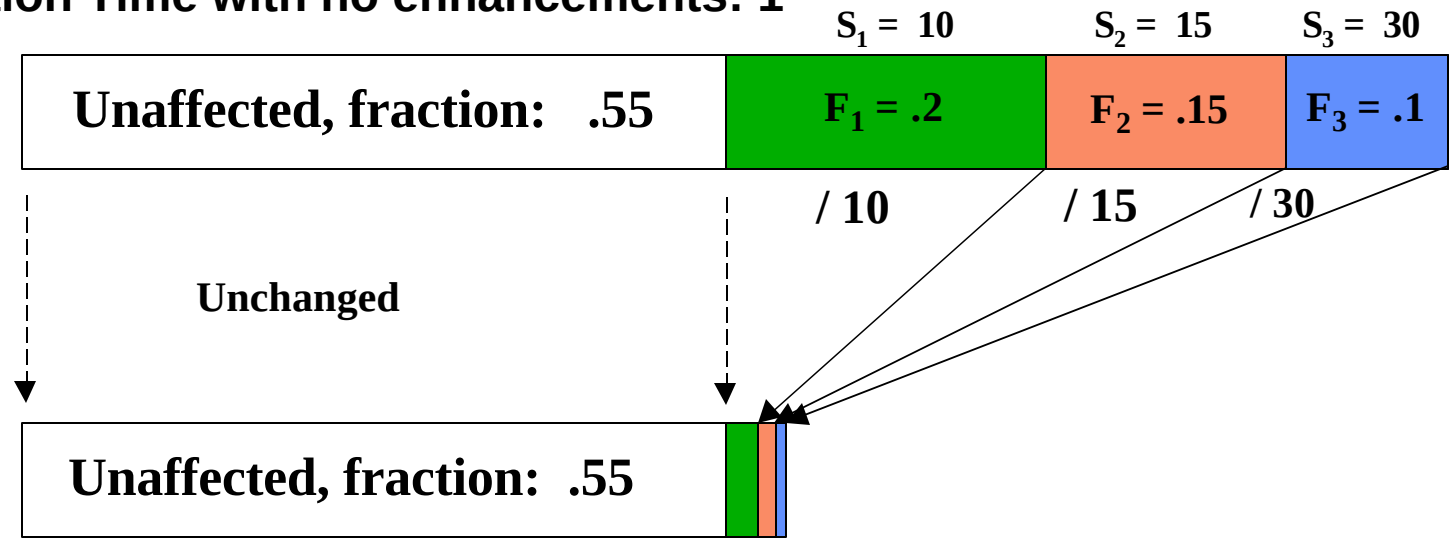
- **While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.**
- **What is the resulting overall speedup?**

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

- **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

Note: All fractions refer to original execution time.

Major CPU Design Steps

1 **Using independent RTN, write the micro-operations required for all target ISA instructions.**

2 **Construct the datapath required by the micro-operations identified in step 1.**

3 **Identify and define the function of all control signals needed by the datapath.**

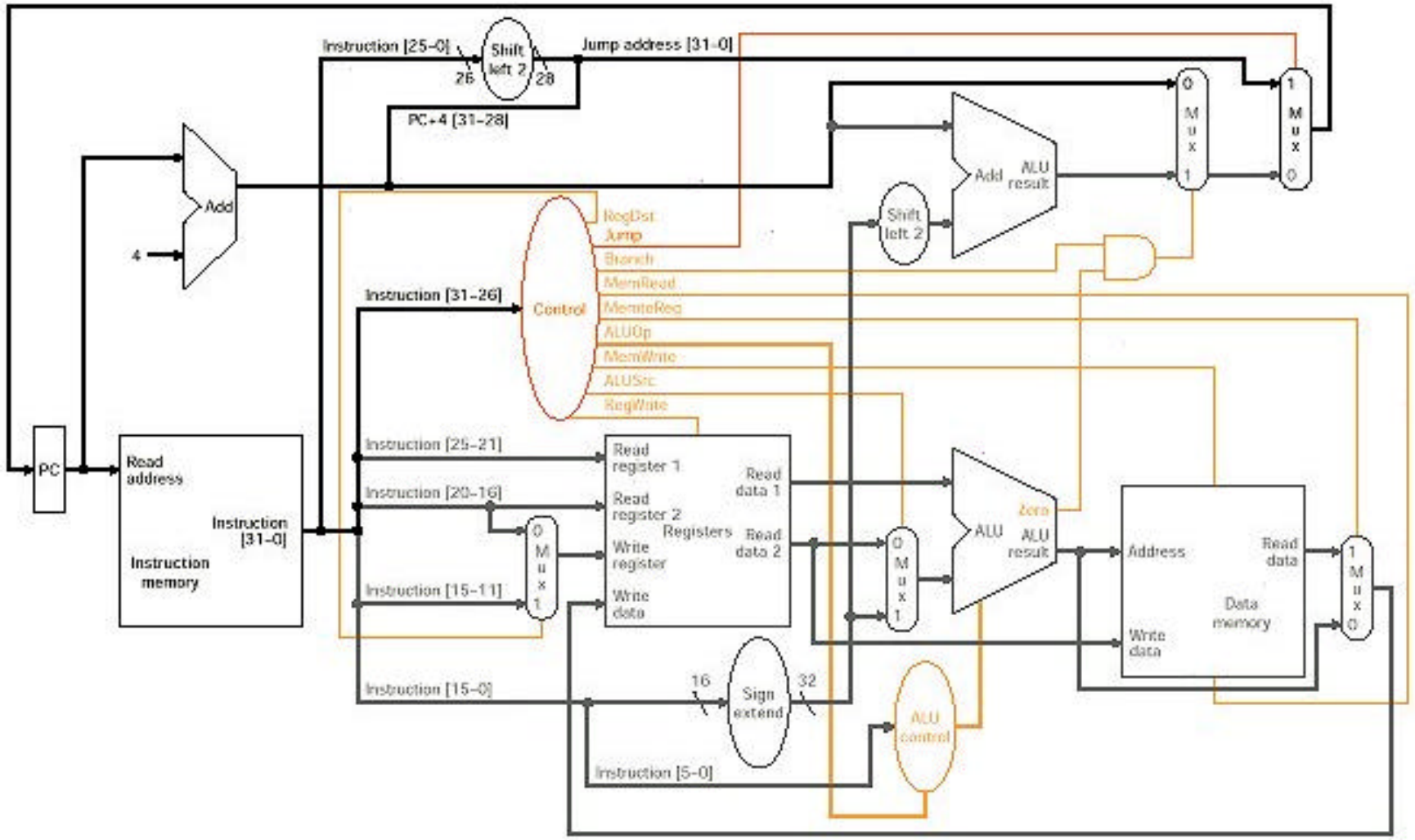
3 **Control unit design, based on micro-operation timing and control signals identified:**

- **Hard-Wired: Finite-state machine implementation**
- **Microprogrammed.**

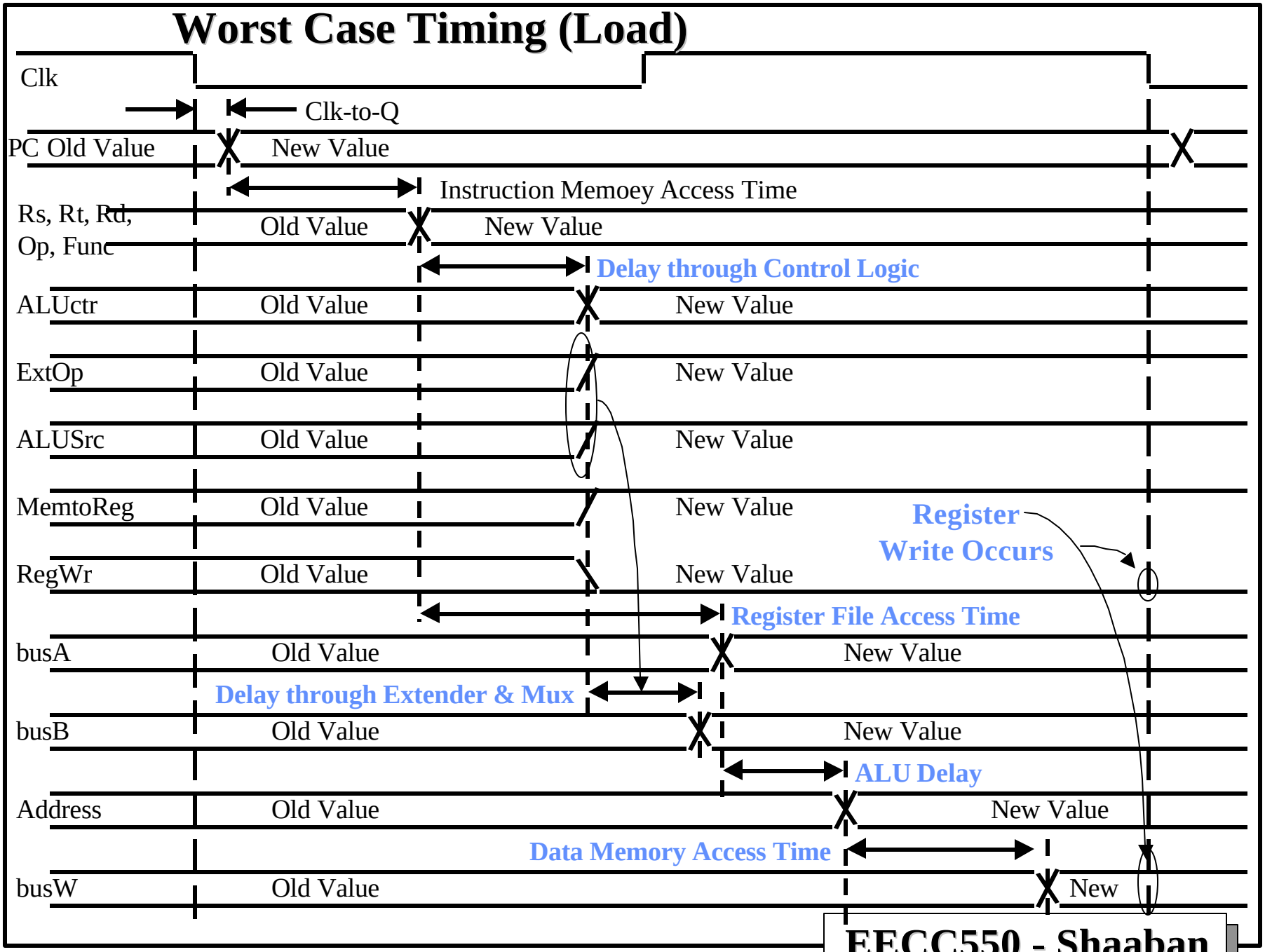
Datapath Design Steps

- **Write the micro-operation sequences required for a number of representative instructions using independent RTN.**
- **From the above, create an initial datapath by determining possible destinations for each data source (i.e registers, ALU).**
  - **This establishes the connectivity requirements (data paths, or connections) for datapath components.**
  - **Whenever multiple sources are connected to a single input, a multiplexer of appropriate size is added.**
- **Find the worst-time propagation delay in the datapath to determine the datapath clock cycle.**
- **Complete the micro-operation sequences for all remaining instructions adding connections/multiplexers as needed.**

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

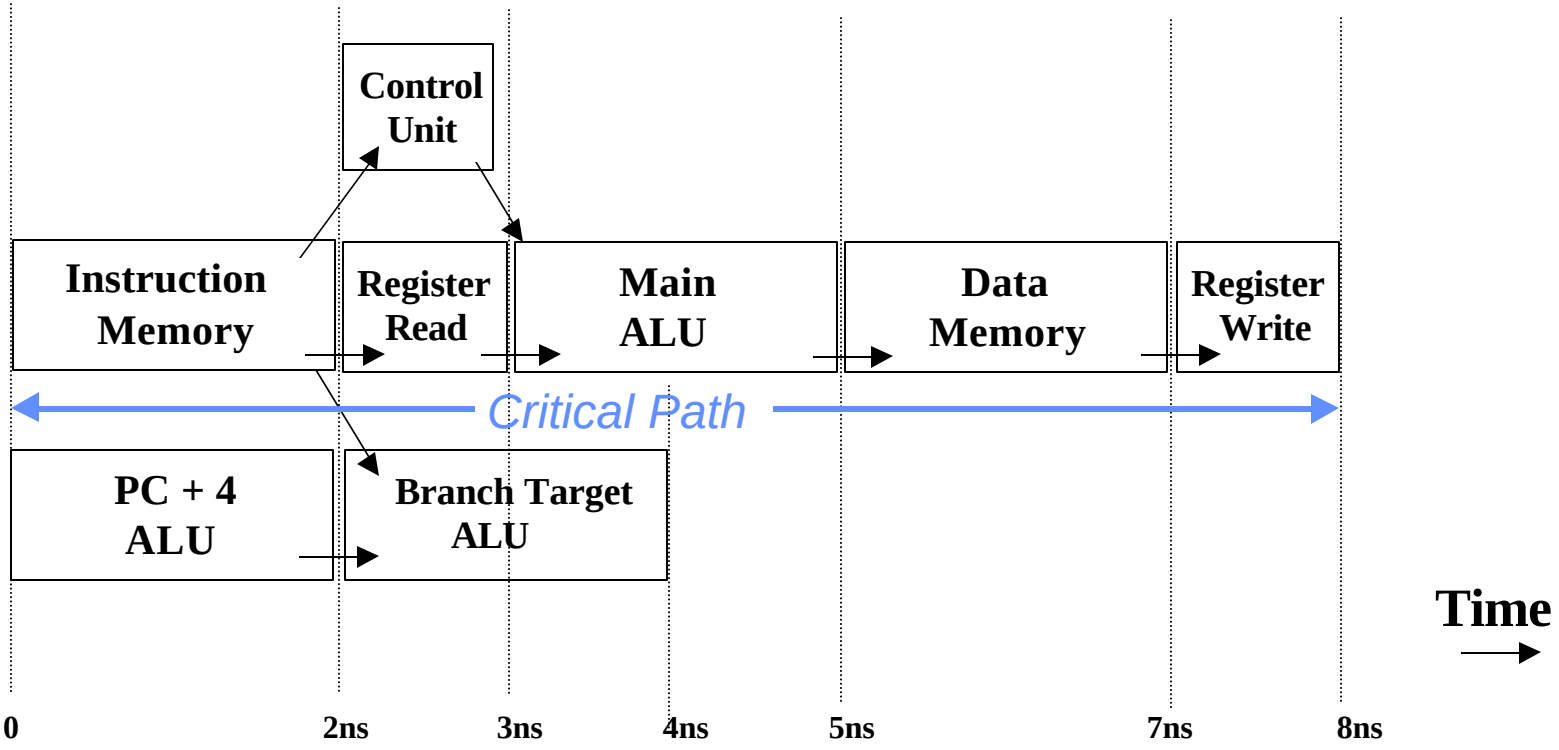


Worst Case Timing (Load)



Simplified Single Cycle Datapath Timing

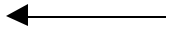
- **Assuming the following datapath/control hardware components delays:**
  - **Memory Units: 2 ns**
  - **ALU and adders: 2 ns**
  - **Register File: 1 ns**
  - **Control Unit < 1 ns**
- **Ignoring Mux and clk-to-Q delays, critical path analysis:**



Performance of Single-Cycle CPU

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

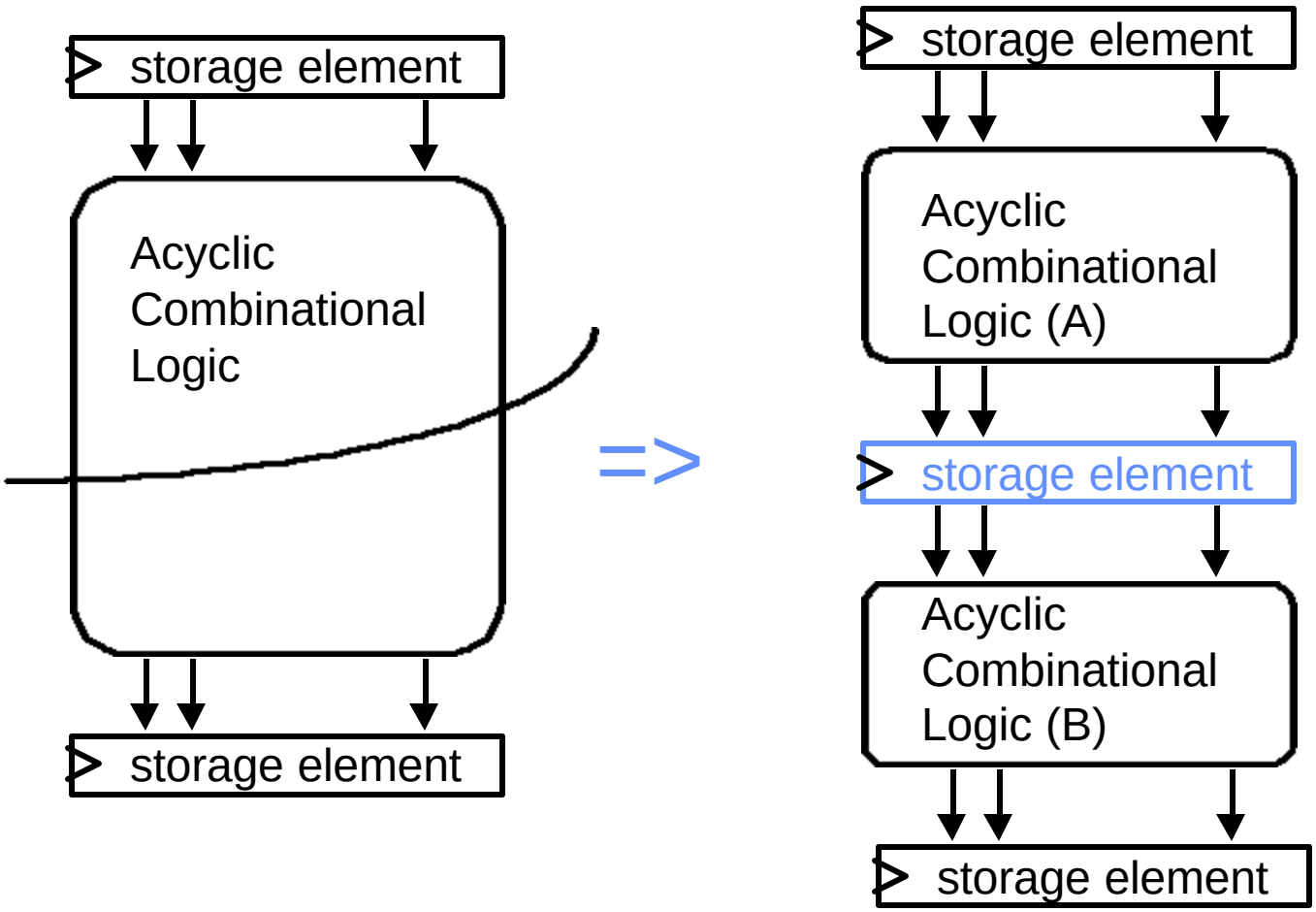


- **The clock cycle is determined by the instruction with longest delay: The load in this case which is 8 ns. Clock rate = 1 / 8 ns = 125 MHz**
- **A program with 1,000,000 instructions takes:**

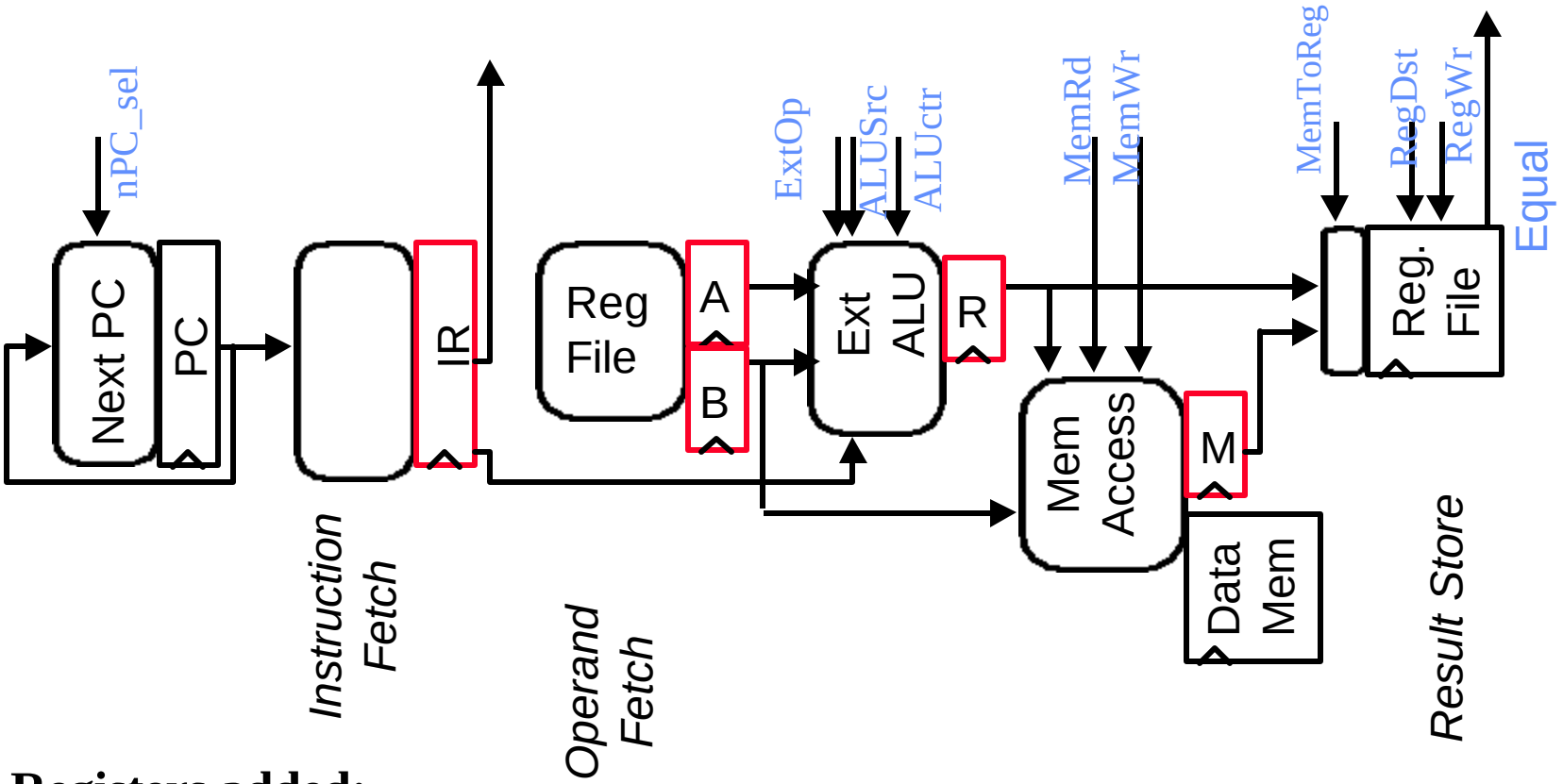
Execution Time = T = I x CPI x C = 10^6 x 1 x $8x10^{-9}$ = 0.008 s = 8 msec

Reducing Cycle Time: Multi-Cycle Design

- **Cut combinational dependency graph by inserting registers / latches.**
- **The same work is done in two or more fast cycles, rather than one slow cycle.**



Example Multi-cycle Datapath



Registers added:

IR: **Instruction register**

A, B: **Two registers to hold operands read from register file.**

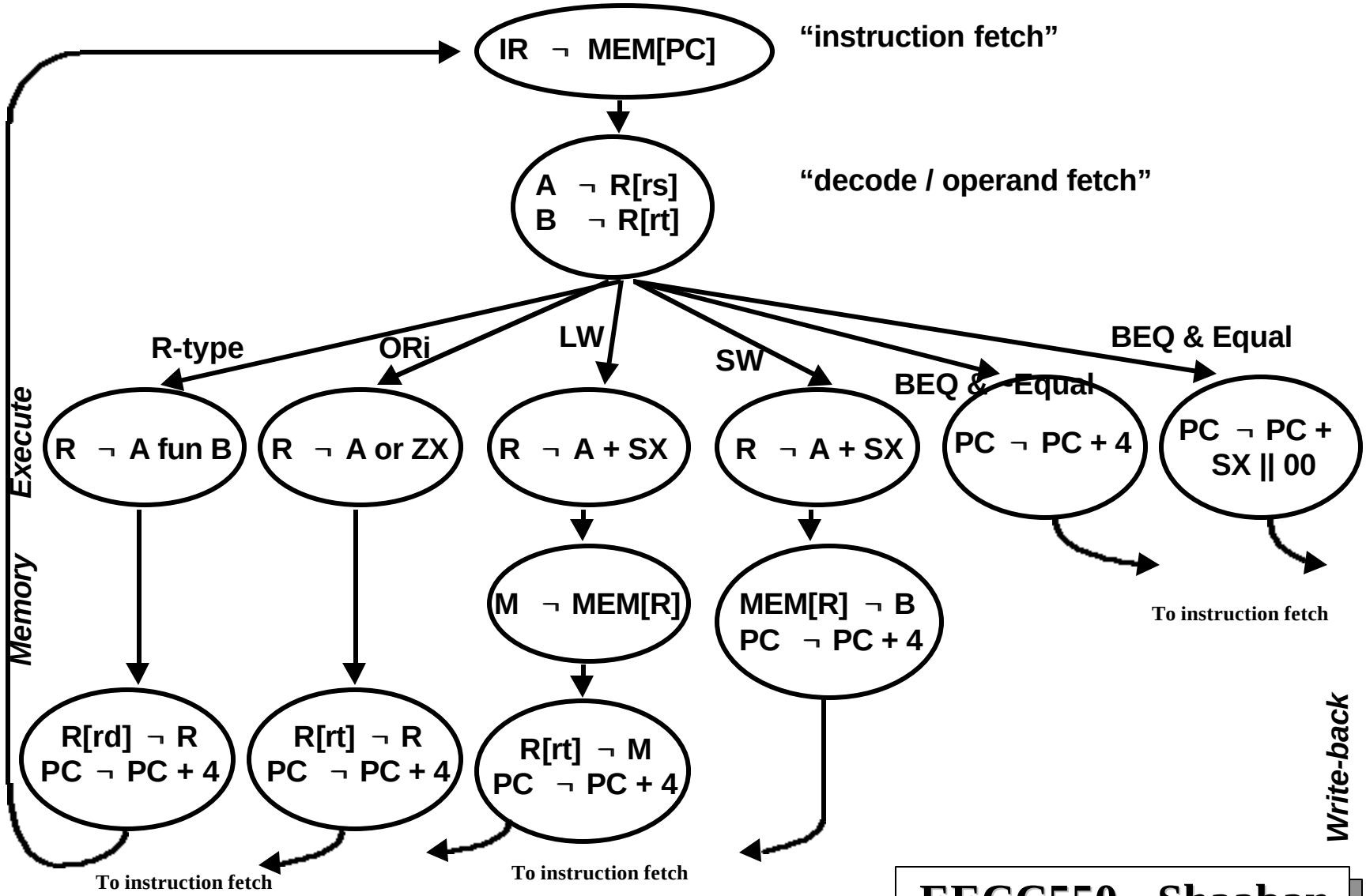
R: **or ALUOut, holds the output of the ALU**

M: **or Memory data register (MDR) to hold data read from data memory**

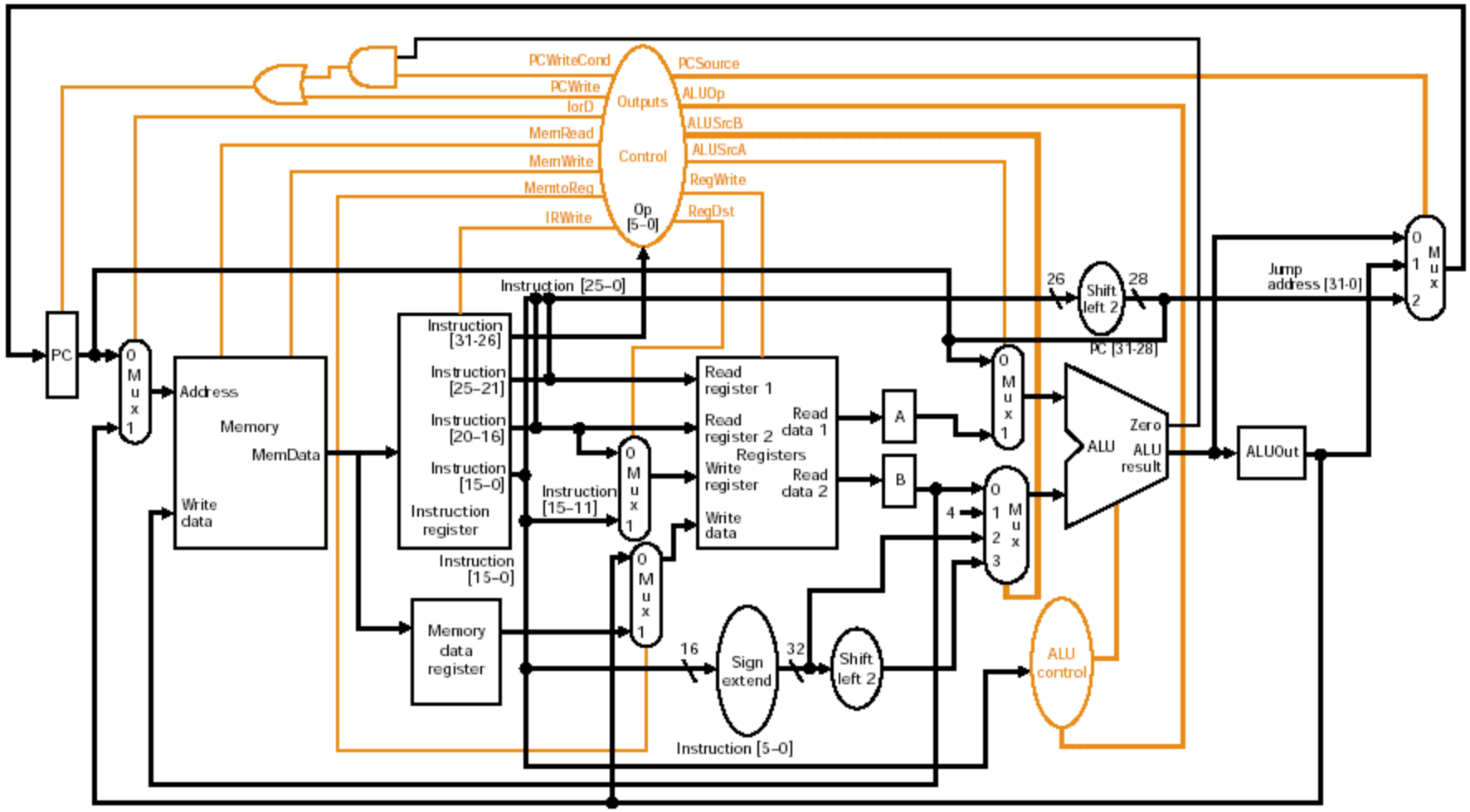
Operations In Each Cycle

	R-Type	Logic Immediate	Load	Store	Branch
Instruction Fetch	IR ← Mem[PC]	IR ← Mem[PC]	IR ← Mem[PC]	IR ← Mem[PC]	IR ← Mem[PC]
Instruction Decode	A ← R[rs] B ← R[rt]	A ← R[rs]	A ← R[rs]	A ← R[rs] B ← R[rt]	A ← R[rs] B ← R[rt]
Execution	R ← A + B	R ← A OR ZeroExt[imm16]	R ← A + SignEx(Im16)	R ← A + SignEx(Im16)	If Equal = 1 PC ← PC + 4 + (SignExt(imm16) x4) else PC ← PC + 4
Memory			M ← Mem[R]	Mem[R] ← B PC ← PC + 4	
Write Back	R[rd] ← R PC ← PC + 4	R[rt] ← R PC ← PC + 4	R[rd] ← M PC ← PC + 4		

Control Specification For Multi-cycle CPU Finite State Machine (FSM)

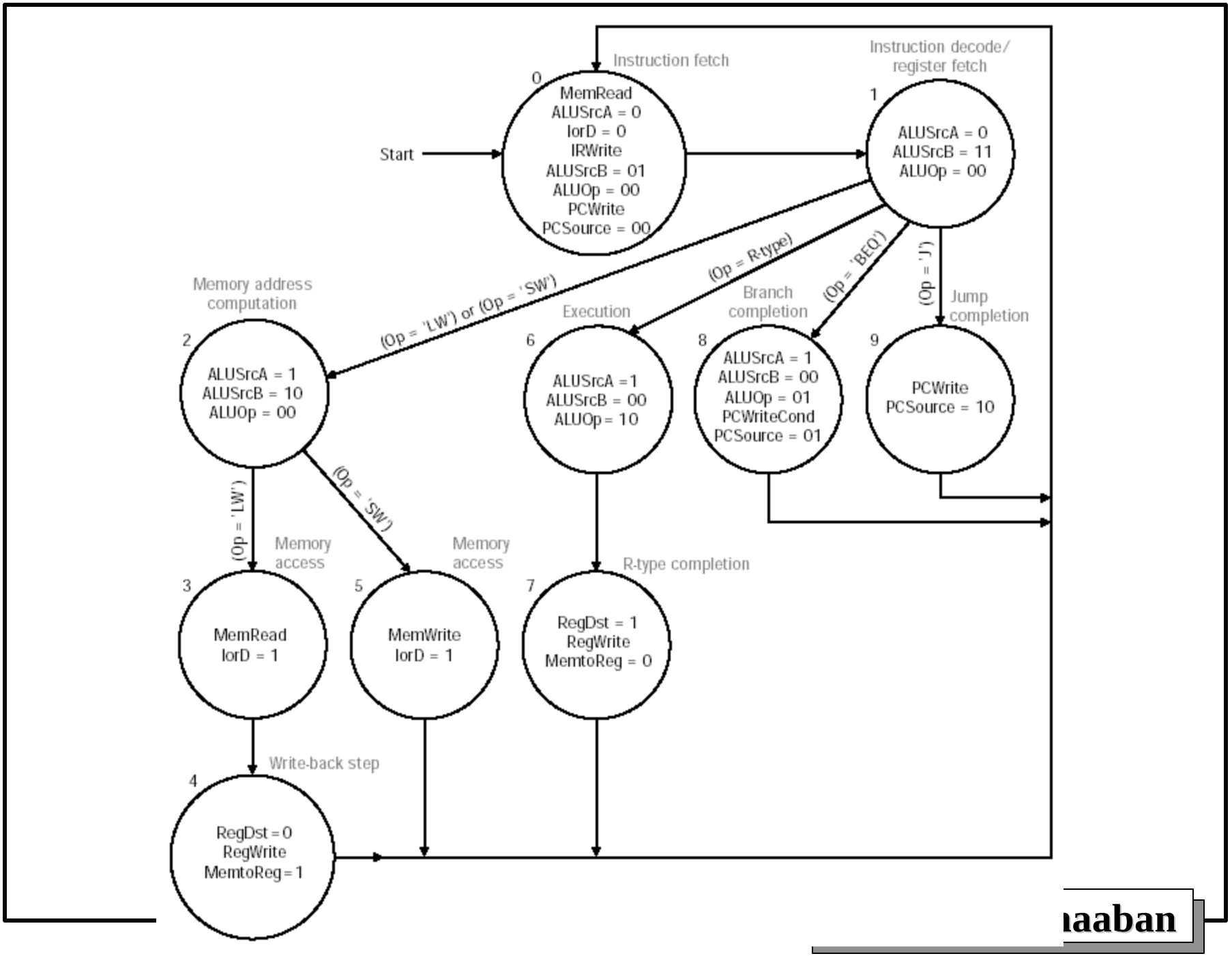


Alternative Multiple Cycle Datapath With Control Lines (Fig 5.33 In Textbook)



Operations In Each Cycle

	R-Type	Logic Immediate	Load	Store	Branch
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
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)
Execution	ALUout ← A + B	ALUout ← A OR ZeroExt[imm16]	ALUout ← A + SignEx(Im16)	ALUout ← A + SignEx(Im16)	If Equal = 1 PC ← ALUout
Memory			M ← Mem[ALUout]	Mem[ALUout] ← B	
Write Back	R[rd] ← ALUout	R[rt] ← ALUout	R[rd] ← Mem		


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA =1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1

MIPS Multi-cycle Datapath Performance Evaluation

- **What is the average CPI?**
  - **State diagram gives CPI for each instruction type**
  - **Workload 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).