CS/EE 320 Computer Organization and Assembly Language Spring 2024

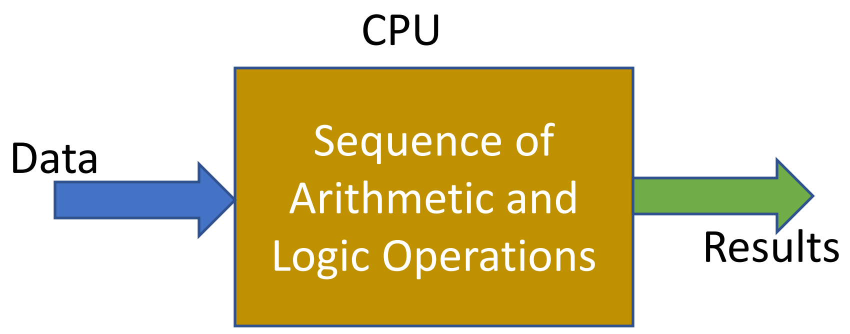
Lecture 5

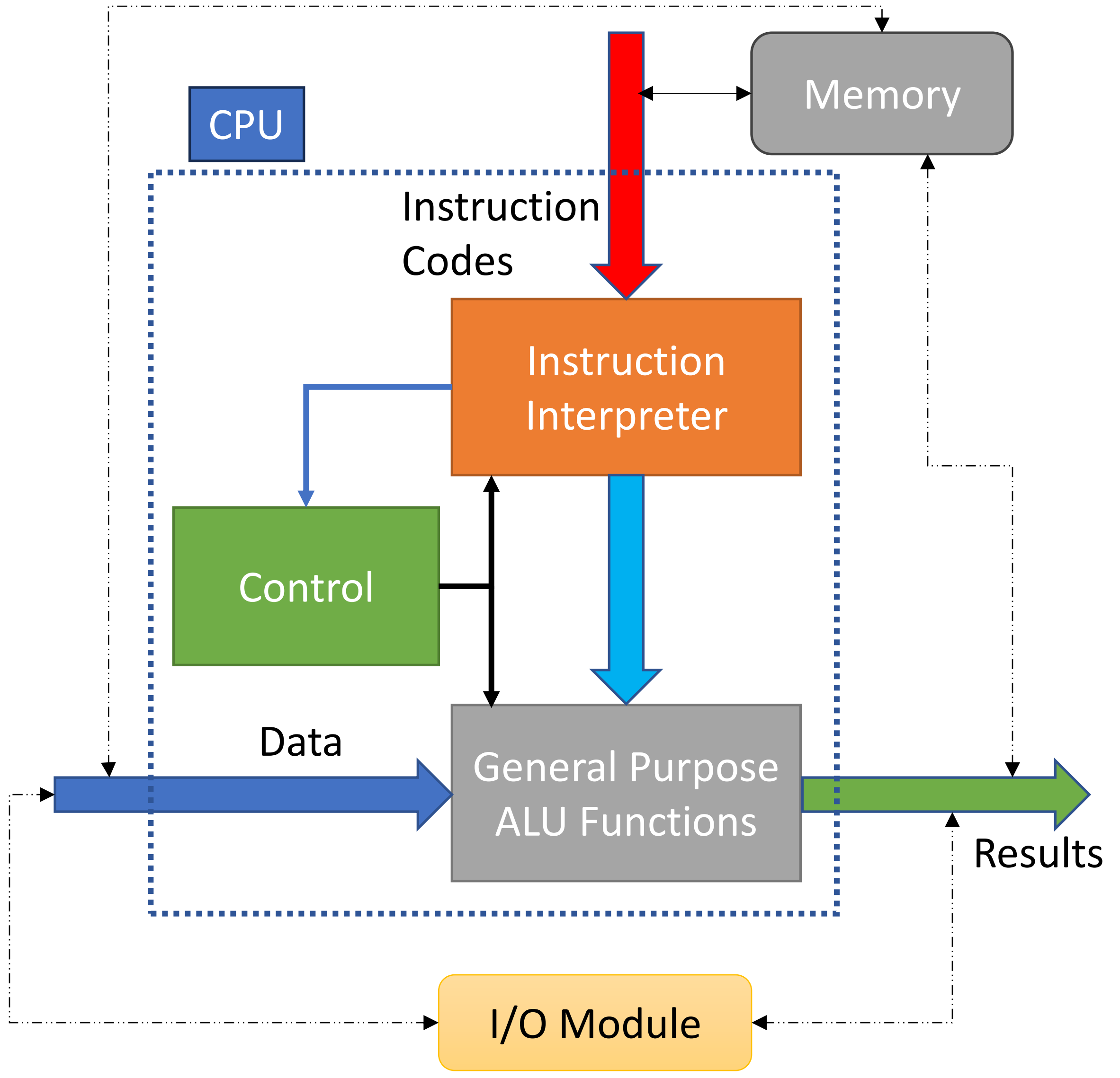
Shahid Masud

Topics

- Basic Working of Von-Neumann Stored Program Computer
- Registers Available Inside CPU: **MBR, MAR, IR, AC, PC**, etc.
- Instruction Execution Cycle Simple – Fetch, Decode, Execute
- Detailed Instruction Execution Cycle with Operand Fetch and Storage of Results, Sequential Processing
- Introducing Assembly Language Instructions
- Identify Assembly Language Instruction from Machine code
- Register files and Memory Access
- Addressing Constraints in R and I Type MIPS Instructions
- Calculating Addresses in Different types of Instructions
- QUIZ 1 next lecture

Basic Computer Operations

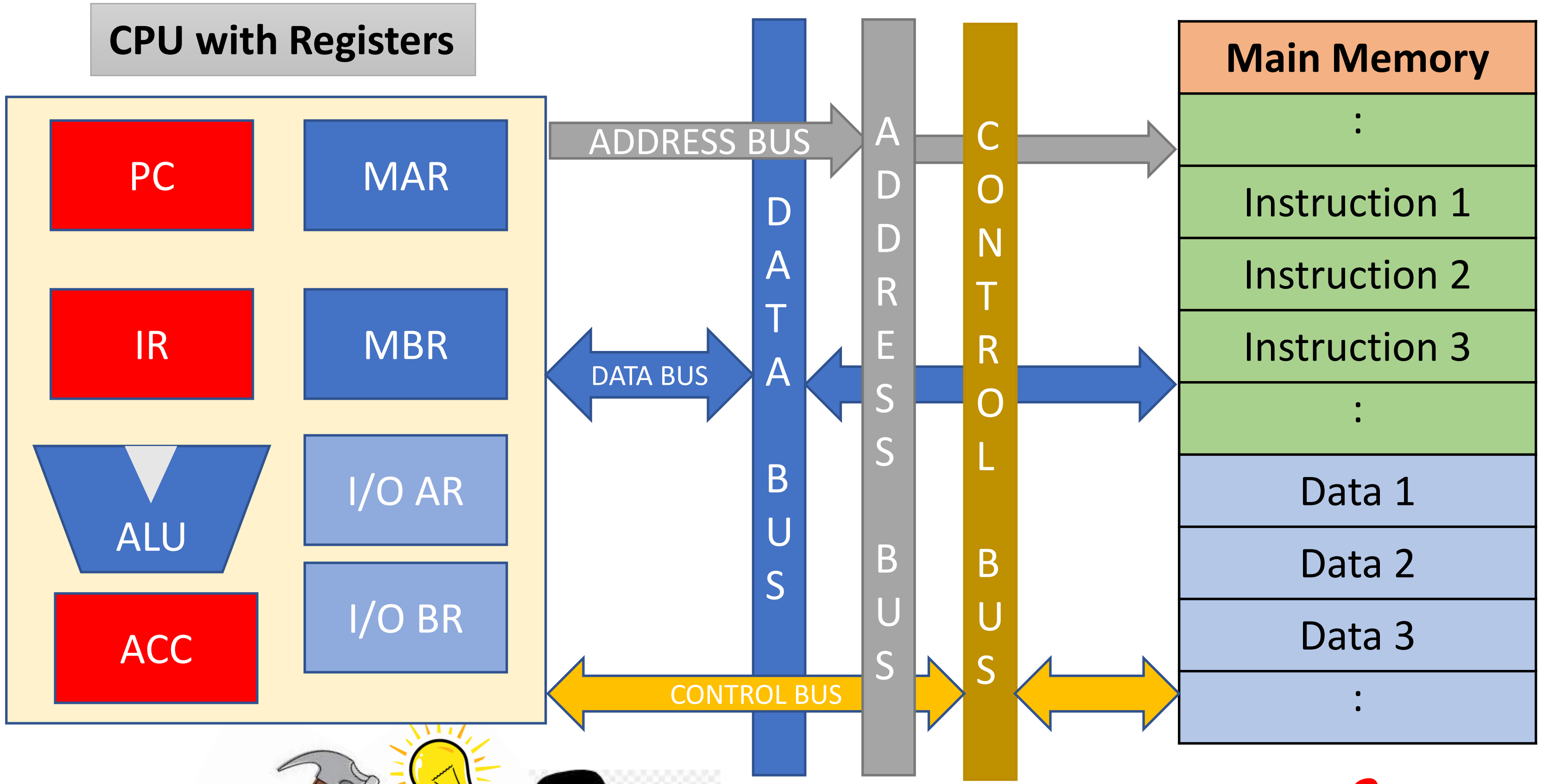




Features of Von-Neuman Architecture

- Data and Instructions are stored in a single read / write Memory
- Contents of Memory are Addressable by Location Address irrespective of stored Contents
- Execution of program occurs in a Sequential Fashion unless Modified through Branch instructions

Essential Registers in a CPU



REGISTERS IN CPU	
ABBREVIATION	**FULL NAME**
PC	Program Counter
IR	Instruction Register
ALU	Arithmetic Logic Unit
ACC	Accumulator
MAR	Memory Address Register
MBR	Memory Buffer Register
I/O AR	I/O Address Register
I/O BR	I/O Buffer Register

Purpose of CPU registers?

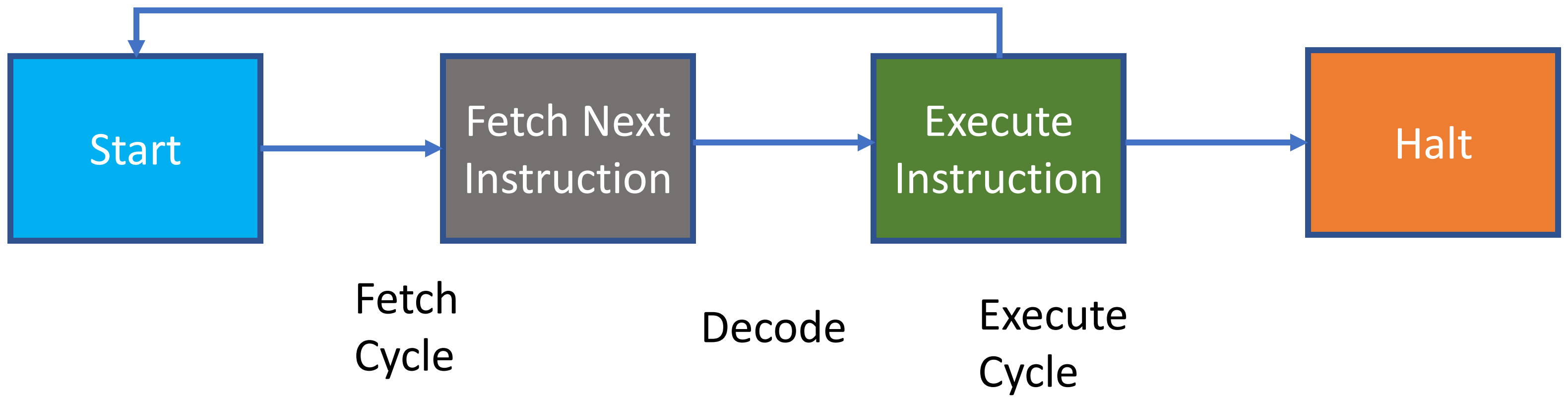
PC, IR and AC Registers

- PC (Program Counter) Register always holds the value of next instruction to be fetched
- PC is incremented by ‘1’ address after each instruction to point to next location in instruction memory
- To alter the sequence of operation, a new value is loaded into the PC
- IR (Instruction Register) Register holds the fetched instruction
- The contents of IR are examined to determine the ALU and Control operation required from this instruction
- AC (Accumulator) register stores results from ALU or other operations inside CPU

Typical Actions by CPU

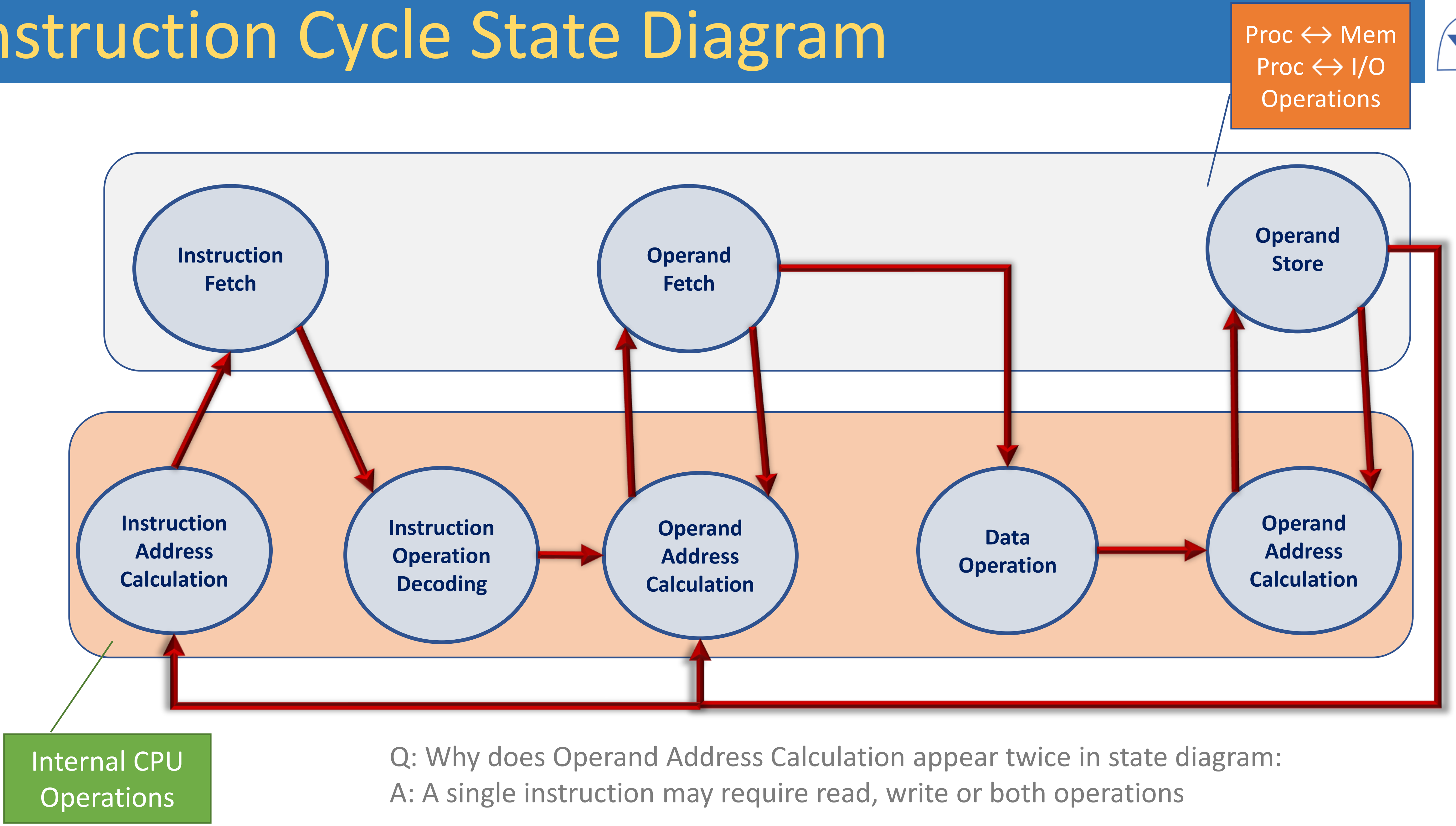
- CPU Actions fall in four categories:
  - Processor – Memory data movement
  - Processor – I/O data movement
  - Data Processing using ALU
  - Control operation to alter the sequence of program execution

Basic Instruction Cycle



- The instruction fetch and execute is repeated until the end of program
- Execution may involve several operations depending upon nature of instruction

Instruction Cycle State Diagram


Proc ↔ Mem
Proc ↔ I/O
Operations
Instruction Fetch
Operand Fetch
Operand Store
Instruction Address Calculation
Instruction Operation Decoding
Operand Address Calculation
Data Operation
Operand Address Calculation
Internal CPU Operations

Q: Why does Operand Address Calculation appear twice in state diagram:
A: A single instruction may require read, write or both operations

Instruction Execution Steps

- PC gives address of instr to be fetched from memory.
- After fetching, the instruction op code is decoded. The processor identifies number of operations. If operand is needed from memory, then its address is calculated.
- Operand fetching process is repeated until all operands are fetched from memory.
- Data operation is performed in ALU and result is produced in ACC.
- If the result is stored in a register than instruction ends here.
- If the destination of result is in memory then destination address is calculated and result moved to memory.
- In tandem, the PC is incremented by ‘1’ **(or 4?)** to determine address of next instruction.
- Instruction cycle is repeated for further instructions.

An Example of Program Execution Cycle

Sequence of Operation

Opcode determines that

AC is to be loaded from memory

Then ADD instruction

Then AC stored in location 941

? CPU needs more than one Register to store second operand

Step 1

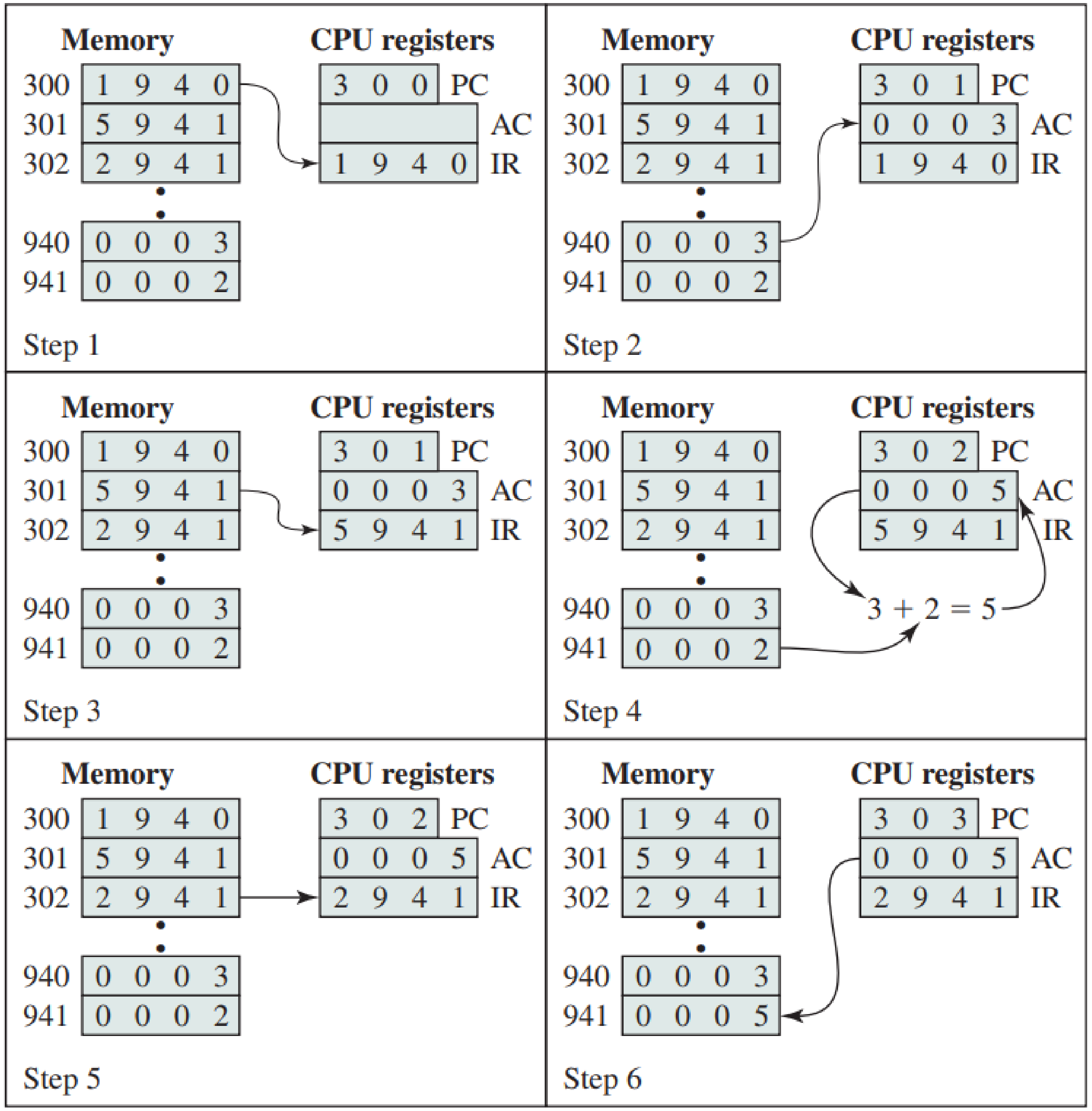
Step 2

Step 3

Step 4

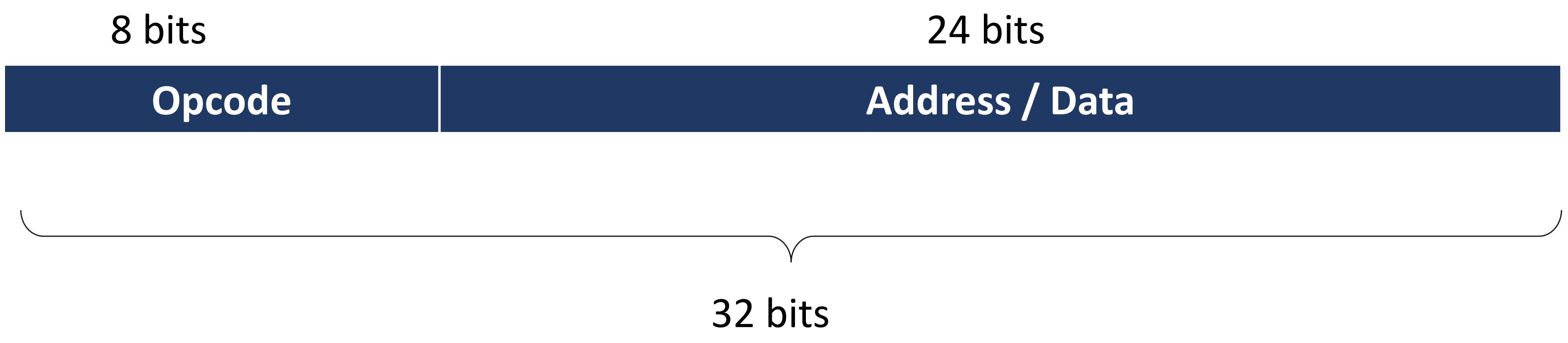
Step 5

Step 6



How is an Assembly Instruction Formed?

As an example:



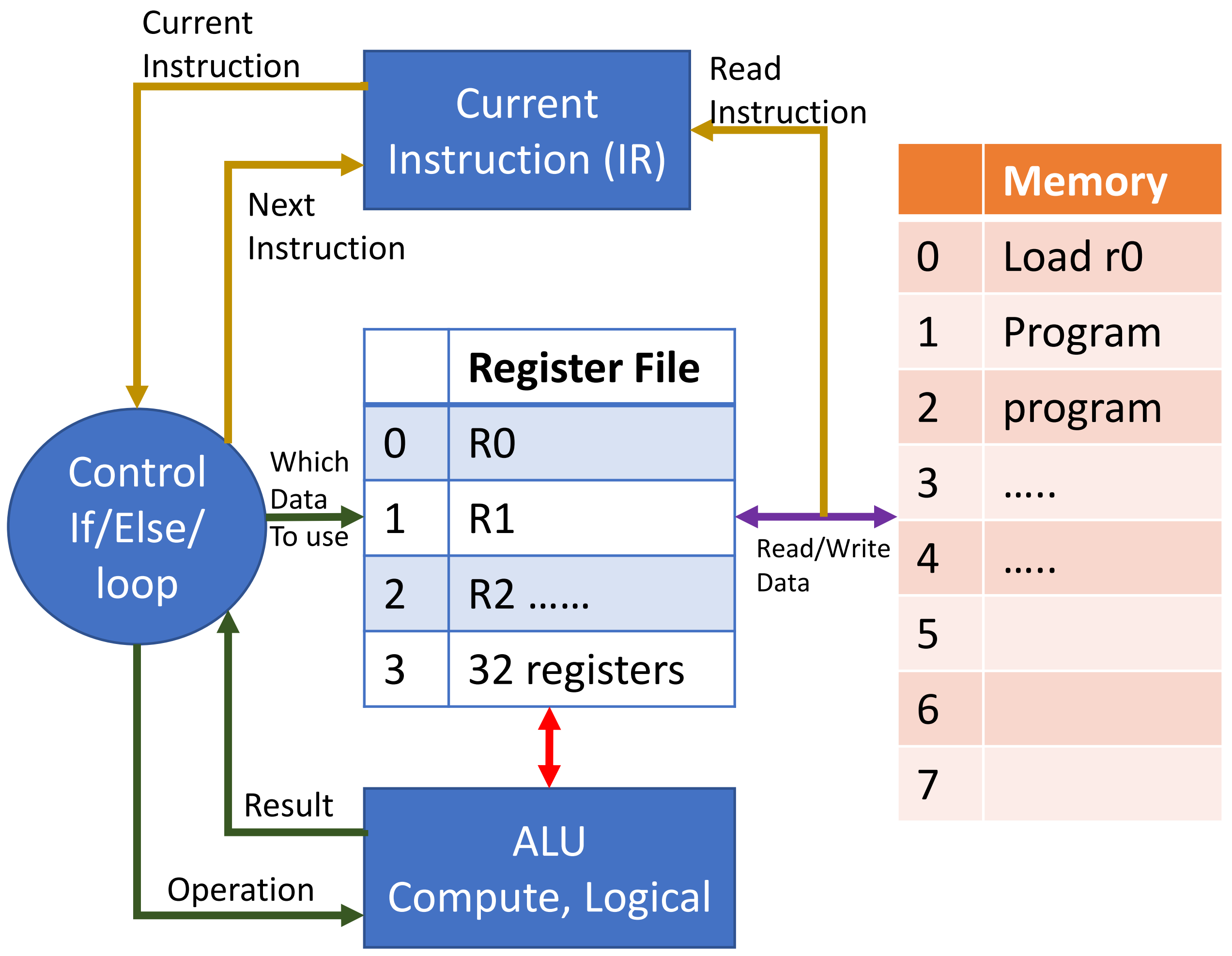
What happens with the full range of Address or Data ?

Answer: Different Addressing Modes

Walking through Program Execution

What does the processor do?

1. Load the instruction from memory
2. Determine what operation to perform
3. Find out where the data is located
4. Perform the operation
5. Determine the next instruction
6. Repeat this process over and over



Register Convention in MIPS Assembly

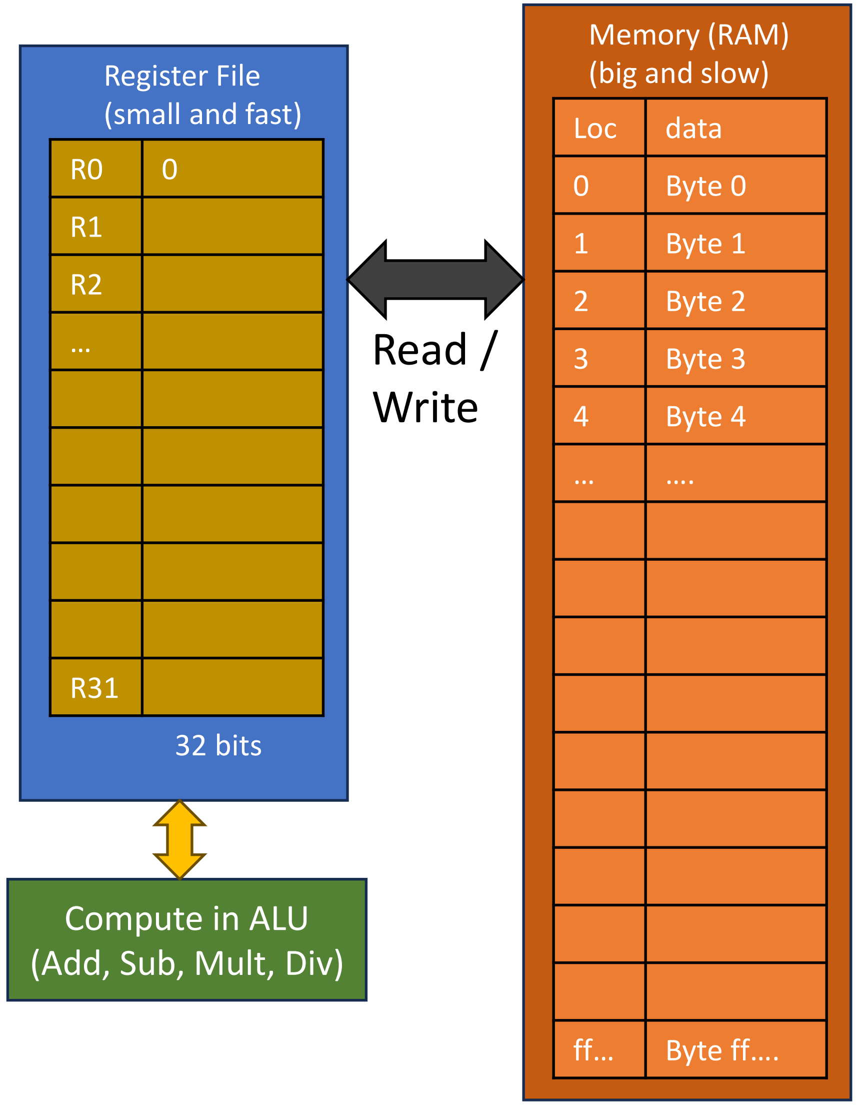
Name	Register No.	Usage
$zero	0	Constant Value 0
$v0 - $v1	2 – 3	Values for results and expression evaluation
$a0 - $a3	4 – 7	arguments
$t0 - $t7	8 – 15	Temporary storage
$s0 – Ss7	16 – 23	Saved
$t8 - $t9	24 – 25	More temporary storage
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address
$at	1	Reserved for Assembler use

Few Special Registers:
- PC (Program Counter)
- Hi and Lo results of Multiplication
- FP Floating Point
- Control Registers for Error and Exception Status

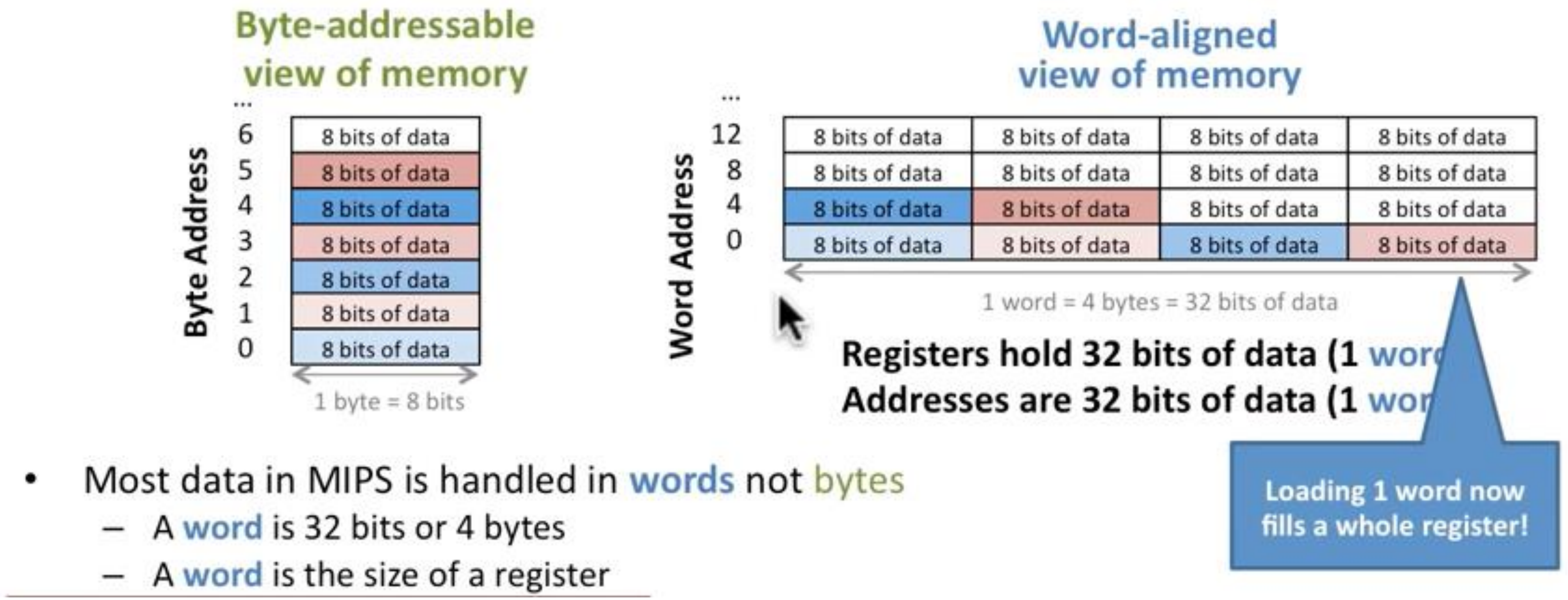
Load / Store Architecture

- MIPS is a Load/Store Register File machine
  - Instructions compute only on data in the Register File
  - Example:
    - **add R3, R2, R1**
    - all data needs to be in the Register File
  - But we only have 32 registers in the Register File
    - Clearly not enough for a big program

- Most data is stored in **memory** (large, but slow)
- Need to transfer the data to the Register File to use it
  - Load: load data from memory to the Register File (lw instruction)
  - Store: store data to the memory from the Register File (sw instruction)



Word Aligned Memory Access

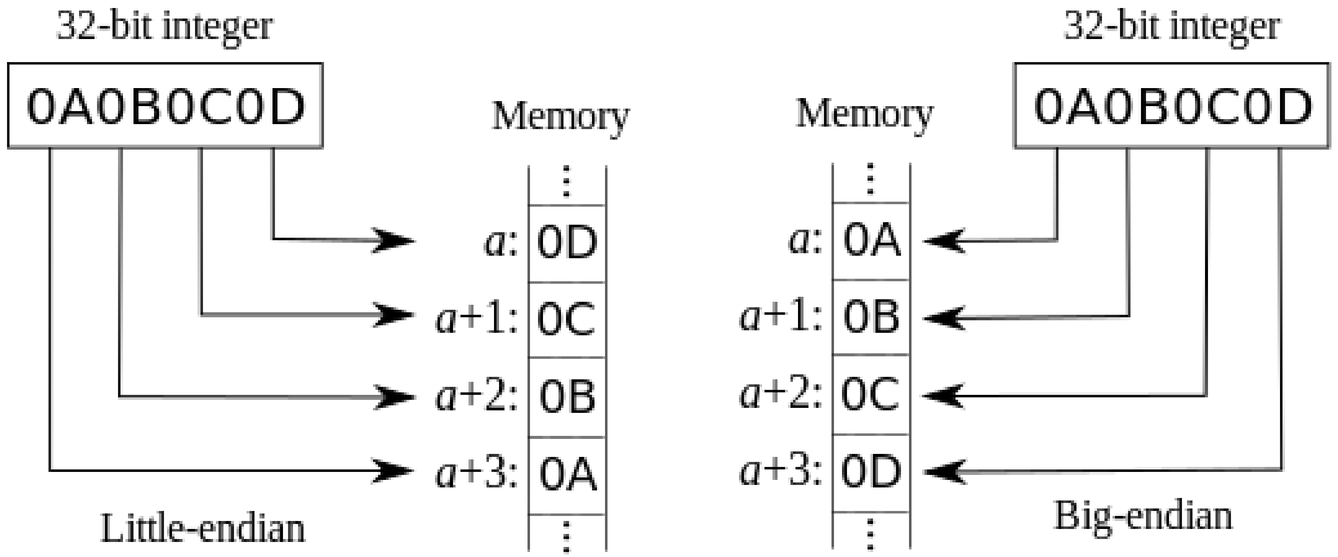

Byte-addressable view of memory
Byte Address
...
6
5
4
3
2
1
0
8 bits of data
1 byte = 8 bits
Word-aligned view of memory
Word Address
...
12
8
4
0
8 bits of data
1 word = 4 bytes = 32 bits of data
Loading 1 word now fills a whole register!

Registers hold 32 bits of data (1 word
Addresses are 32 bits of data (1 wor

- Most data in MIPS is handled in words not bytes
  - A word is 32 bits or 4 bytes
  - A word is the size of a register

What are last 2 bits of Word Addresses?

Memory Address in Big Endian and Little Endian



Register File and Memory Access in MIPS

- MIPS is Characterized by **Load Store Architecture**
- MIPS has a **Word Aligned Memory Access**
- MIPS has a **‘Big Endian’** Register to Memory transfer

Starting with MIPS Instructions

- General 3-operand format:
  - *op* dest, src1, src2 $\boxed{\text{dest} \leftarrow \text{src1 } op \text{ src2}}$

dest, src1, src2 are registers

- Addition
  - add a, b, c a ← b + c
  - addi a, b, 12 a ← b + 12

The "i" in addi is for immediate

- Subtraction
  - sub a, b, c a ← b - c

- Complex: $f = \underline{(g + h)} - \underline{(i + j)}$
  - add t0, g, h $\underline{t0} \leftarrow g + h$
  - add t1, i, j $\underline{t1} \leftarrow i + j$
  - sub f, t0, t1 $f \leftarrow \underline{t0} - \underline{t1}$

Complex operations generate many instructions with temporary values.

Three main types of MIPS Instructions

Data Operations

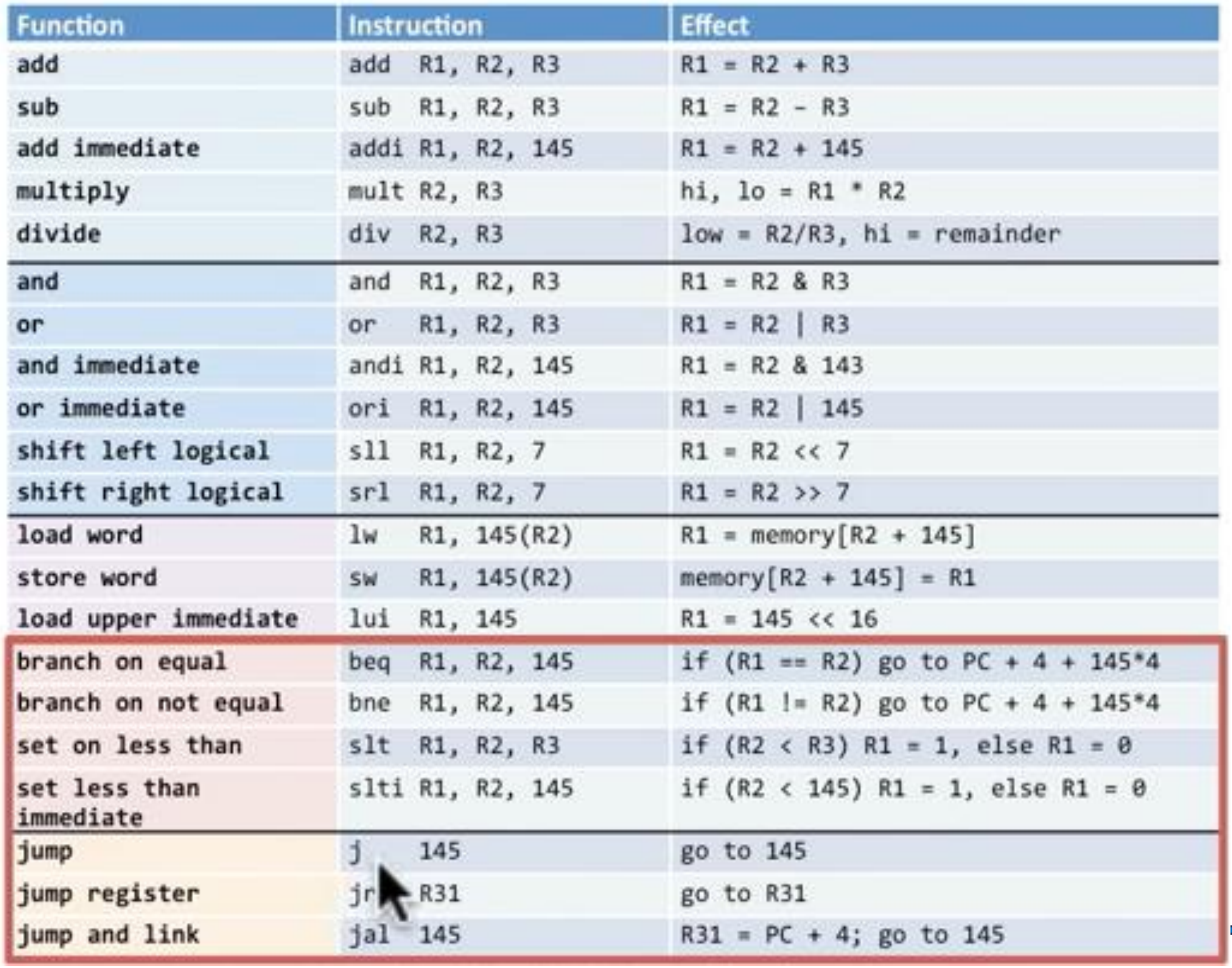
- Arithmetic (add, sub, …)
- Logical (and, nor, xor, …)

Data Transfer

- Load (mem to reg, …)
- Store (reg to mem, …)

Sequencing

- Branch (conditional, =0, …)
- Jump (unconditional, …)



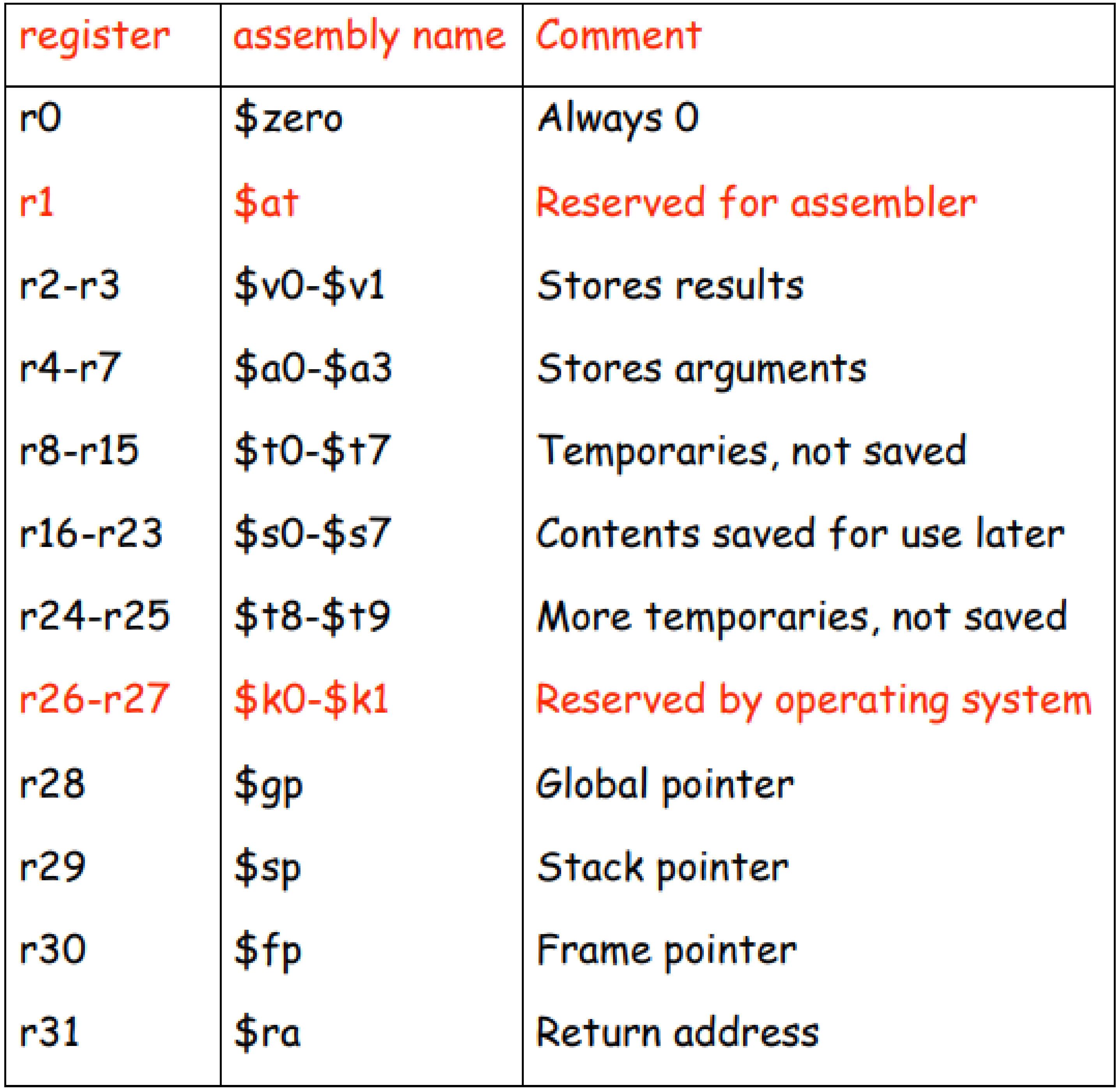
Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 - R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R1 * R2
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 143
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

MIPS R Format Instructions

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

- **Instruction fields**
  - **op**: operation code (opcode)
  - **rs**: first source register number
  - **rt**: second source register number
  - **rd**: destination register number
  - **shamt**: shift amount
  - **funct**: function code (extends opcode)

Register numbering for R type instructions



register	assembly name	Comment
r0	$zero	Always 0
r1	$at	Reserved for assembler
r2-r3	$v0-$v1	Stores results
r4-r7	$a0-$a3	Stores arguments
r8-r15	$t0-$t7	Temporaries, not saved
r16-r23	$s0-$s7	Contents saved for use later
r24-r25	$t8-$t9	More temporaries, not saved
r26-r27	$k0-$k1	Reserved by operating system
r28	$gp	Global pointer
r29	$sp	Stack pointer
r30	$fp	Frame pointer
r31	$ra	Return address

R Format Instruction Example

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

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Dealing with Constant Values

- E.g., program has variables x=1, y=4, f=3, etc.
- First way: First load all constants into registers:
  - Not so many registers
  - Instructions Cycles are wasted
- Second way: Use ‘i’ instructions such as addi, subi where a constant can be made part of instruction:
- E.g. addi R29, R29, 4
- But R type instructions do not have so many bits in each field
- Solution: Use I type instructions with immediate data value

Some examples of R type instructions from MIPS reference sheet

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - **rt**: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in **rs**

I Type MIPS Instruction Example

Instructions with Immediate Data Type

Example: Determine Machine code for this
Assembly Language Instruction:

lw $t0, 1002($s2) ; load word into $t0

op	rs	rt	16 bit constant offset
100011	10010	01000	0000001111101010

Control module in CPU tells the ALU to take first operand from Register file and the second operand from the Instruction (available in IR register)
The 16 bit Offset in I type instructions can refer to 'Data' or 'Address'. Both are treated separately

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

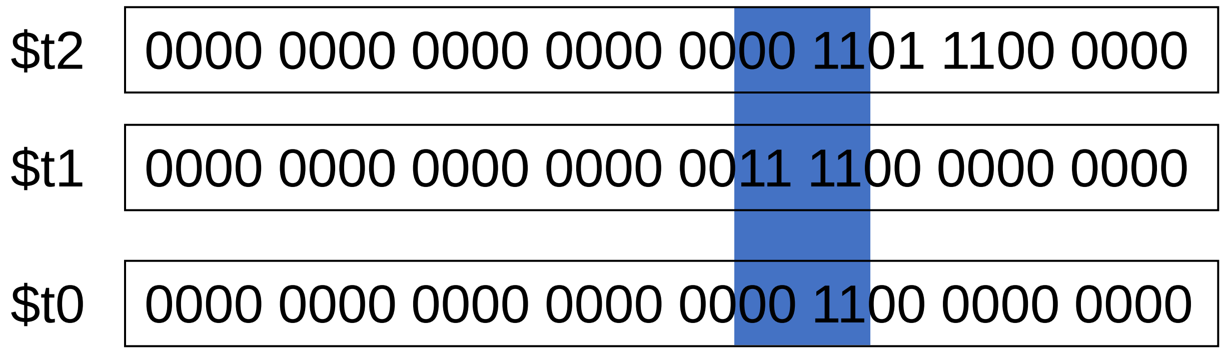
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```



```
$t2   0000 0000 0000 0000 0000 1101 1100 0000
$t1   0000 0000 0000 0000 0011 1100 0000 0000
$t0   0000 0000 0000 0000 0000 1100 0000 0000
```

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

```
$t2   0000 0000 0000 0000 0000 1101 1100 0000

$t1   0000 0000 0000 0000 0011 1100 0000 0000

$t0   0000 0000 0000 0000 0011 1101 1100 0000
```

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Readings

- Chapter 2 of P&H Textbook