Computer Organization and Architecture

Prof. Indranil Sengupta

Dr. Sarani Bhattacharya

Department of Computer Science and Engineering

IIT Kharagpur

MIPS32 INSTRUCTION SET

2

Instruction Set Classification

- MIPS32 instruction can be classified into the following functional groups:
  a) Load and Store
  b) Arithmetic and Logical
  c) Jump and Branch
  d) Miscellaneous
  e) Coprocessor instruction (to activate an auxiliary processor).
- All instructions are encoded in 32 bits.

Alignment of Words in Memory

- MIPS requires that all words must be aligned in memory to word boundaries.
  - Must start from an address that is some power of 4.
  - Last two bits of the address must be 00.
- Allows a word to be fetched in a single cycle.
  - Misaligned words may require two cycles.

Address				
0000H	w1	w1	w1	w1
0004H		w2	w2	w2
0008H	w2			
000CH			w3	w3
0010H	w3	w3		
0014H				w4
0018H	w4	w4	w4	

w1 is aligned, but w2, w3, w4 are not

(a) Load and Store Instructions

- MIPS32 is a load-store architecture.
  - All operations are performed on operands held in processor registers.
  - Main memory is accessed only through *LOAD* and *STORE* instructions.
- There are various types of LOAD and STORE instructions, each used for a particular purpose.
  a) By specifying the size of the operand (W: word, H: half-word, B: byte)
     - Examples: LW, LH, LB, SW, SW, SB
  b) By specifying whether the operand is signed (by default) or unsigned.
     - Examples: LHU, LBU

c) Accessing fields that are not word aligned.
   - Examples: LWL, LWR, SWL, SWR

d) Atomic memory update for read-modify-write instructions
   - Examples: LL, SC

Data sizes that can be accessed through LOAD and STORE

Data Size	Load Signed	Load Unsigned	Store
Byte	YES	YES	YES
Half-word	YES	YES	YES
Word	YES	Only for MIPS64	YES
Unaligned word	YES		YES
Linked word (atomic modify)	YES		YES

Type	Mnemonic	Function
Aligned	LB	Load Byte
	LBU	Load Byte Unsigned
	LH	Load Half-word
	LHU	Load Half-word Unsigned
	LW	Load Word
	SB	Store Byte
	SH	Store Half-word
	SW	Store Word

Type	Mnemonic	Function
Unaligned	LWL	Load Word Left
	LWR	Load Word Right
	SWL	Store Word Left
	SWR	Store Word Right
Atomic Update	LL	Load Linked Word
	SB	Store Conditional Word

(b) Arithmetic and Logic Instructions

- All arithmetic and logic instructions operate on registers.
- Can be broadly classified into the following categories:
  - ALU immediate
  - ALU 3-operand
  - ALU 2-operand
  - Shift
  - Multiply and Divide

Type	Mnemonic	Function
16-bit Immediate Operand	ADDI	Add Immediate Word
	ADDIU	Add Immediate Unsigned Word
	ANDI	AND Immediate
	LUI	Load Upper Immediate
	ORI	OR Immediate
	SLTI	Set on Less Than Immediate
	SLTIU	Set on Less Than Immediate Unsigned
	XORI	Exclusive-OR Immediate

Type	Mnemonic	Function
3-Operand	ADD	Add Word
	ADDU	Add Unsigned Word
	AND	Logical AND
	NOR	Logical NOR
	SLT	Set on Less Than
	SLTU	Set on Less Than Unsigned
	SUB	Subtract Word
	SUBU	Subtract Unsigned Word
	XOR	Logical XOR

Type	Mnemonic	Function
Shift	ROTR	Rotate Word Right
	ROTRV	Rotate Word Right Value (Register)
	SLL	Shift Word Left Logical
	SLLV	Shift Word Left Logical Value (Register)
	SRA	Shift Word Right Arithmetic
	SRAV	Shift Word Right Arithmetic Value (Register)
	SRL	Shift Word Right Logical
	SRLV	Shift Word Right Logical Value (Register)

(c) Multiply and Divide Instructions

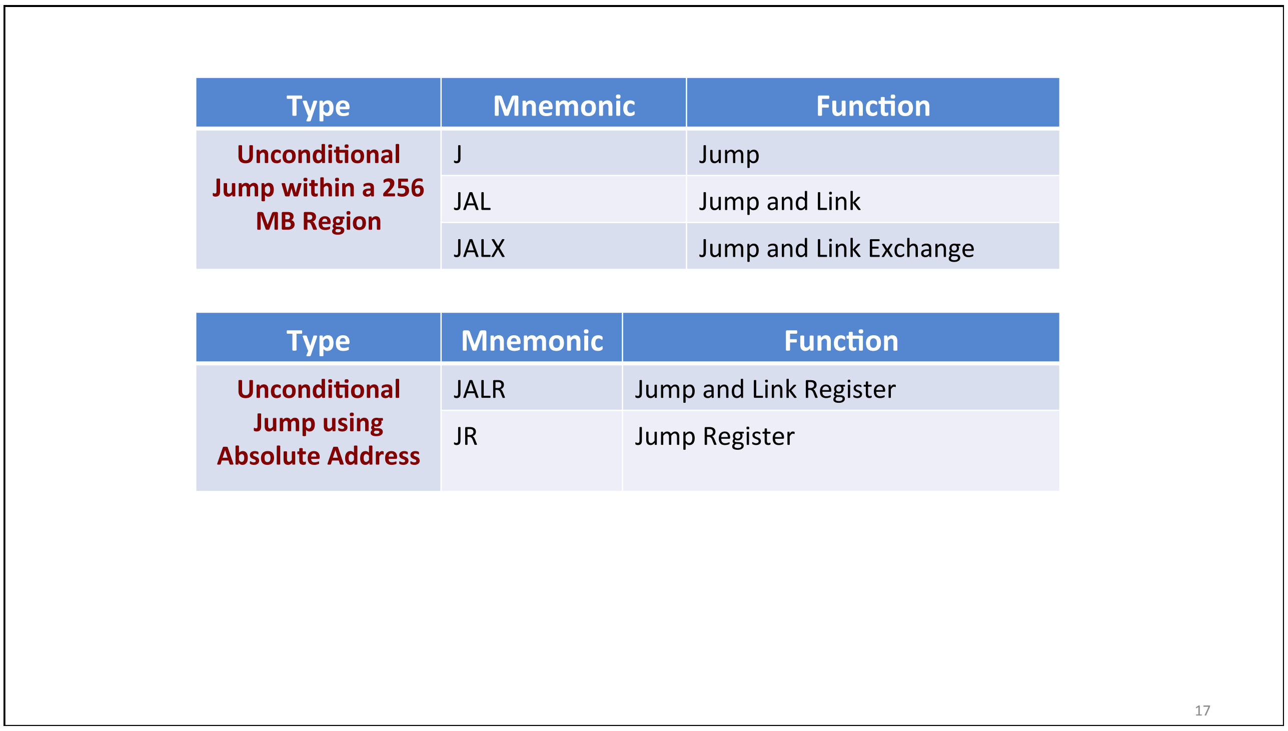
- The multiply and divide instructions produce twice as many result bits.
  - When two 32-bit numbers are multiplied, we get a 64-bit product.
  - After division, we get a 32-bit quotient and a 32-bit remainder.
- Results are produced in the HI and LO register pair.
  - For multiplication, the low half of the product is loaded into LO, while the higher half in HI.
  - Multiply-Add and Multiply-Subtract produce a 64-bit product, and adds or subtracts the product from the concatenated value of HI and LO.
  - Divide produces a quotient that is loaded into LO and a remainder that is loaded into HI.

- Only exception is the MUL instruction, which delivers the lower half of the result directly to a GPR.
  - Useful is situations where the product is expected to fit in 32 bits.

Type	Mnemonic	Function
Multiply and Divide	DIV	Divide Word
	DIVU	Divide Unsigned Word
	MADD	Multiply and Add Word
	MADDU	Multiply and Add Word Unsigned
	MFHI	Move from HI
	MFLO	Move from LO
	MSUB	Multiply and Subtract Word
	MSUBU	Multiply and Subtract Word Unsigned
	MTHI	Move to HI
	MTLO	Move to LO
	MUL	Multiply Word to Register
	MULT	Multiply Word
	MULTU	Multiply Unsigned Word

(d) Jump and Branch Instructions

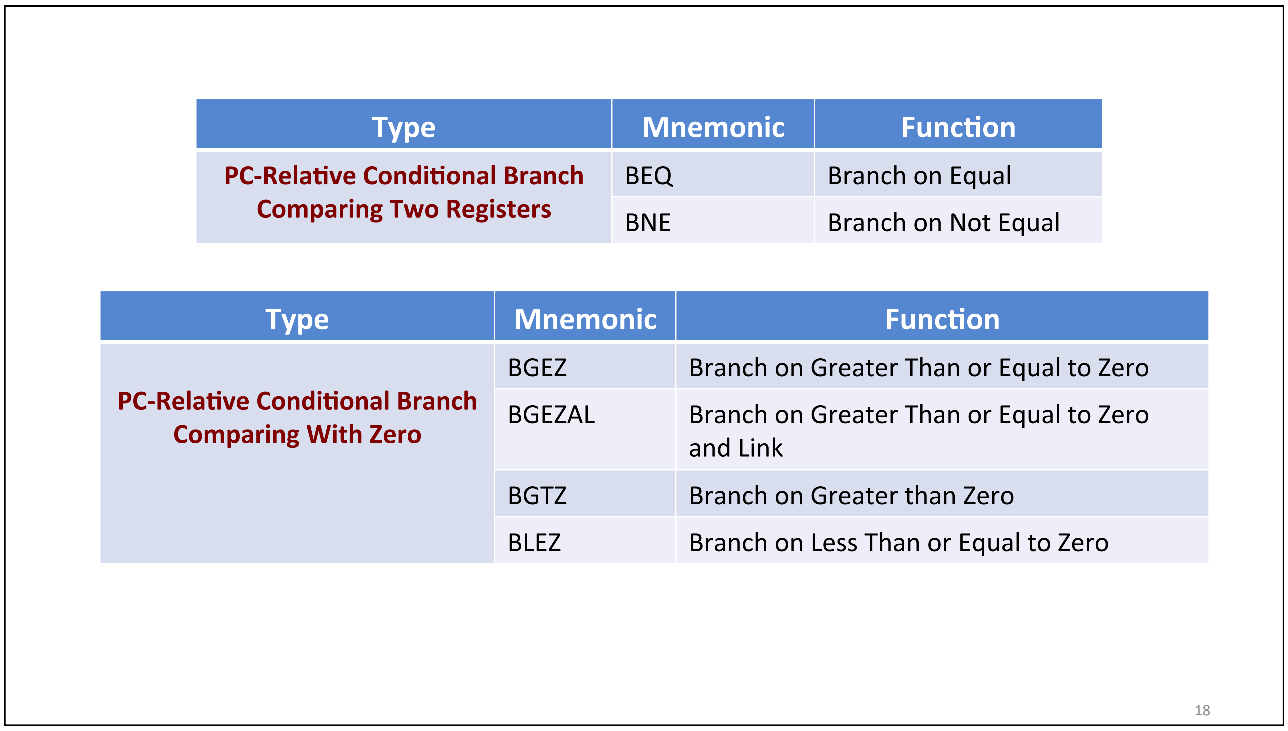
- The following types of Jump and Branch instructions are supported by MIPS32.
  - PC relative conditional branch
    - A 16-bit offset is added to PC.
  - PC-relative unconditional jump
    - A 28-bit offset if added to PC.
  - Absolute (register) unconditional jump
  - Special Jump instructions that link the return address in R31.



Type	Mnemonic	Function
Unconditional Jump within a 256 MB Region	J	Jump
	JAL	Jump and Link
	JALX	Jump and Link Exchange

Type	Mnemonic	Function
Unconditional Jump using Absolute Address	JALR	Jump and Link Register
	JR	Jump Register

17



Type	Mnemonic	Function
PC-Relative Conditional Branch Comparing Two Registers	BEQ	Branch on Equal
	BNE	Branch on Not Equal

Type	Mnemonic	Function
PC-Relative Conditional Branch Comparing With Zero	BGEZ	Branch on Greater Than or Equal to Zero
	BGEZAL	Branch on Greater Than or Equal to Zero and Link
	BGTZ	Branch on Greater than Zero
	BLEZ	Branch on Less Than or Equal to Zero

18

(e) Miscellaneous Instructions

- These instructions are used for various specific machine control purposes.
- They include:
  - Exception instructions
  - Conditional MOVE instructions
  - Prefetch instructions
  - NOP instructions

19

Type	Mnemonic	Function
System Call and Breakpoint	BREAK	
	SYSCALL	

Type	Mnemonic	Function
Trap-on-Condition Comparing Two Registers	TEQ	
	TGE	
	TGEU	
	TLT	Trap if Less Than
	TLTU	Trap if Less Than Unsigned
	TNE	Trap if Not Equal

Type	Mnemonic	Function
Trap-on-Condition Comparing an Immediate Value	TEQI	Trap if Equal Immediate
	TGEI	Trap if Greater Than or Equal Immediate
	TGEIU	Trap if Greater Than or Equal Immediate Unsigned
	TLTI	Trap if Less Than Immediate
	TLTIU	Trap if Less Than Immediate Unsigned
	TNEI	Trap if Not Equal Immediate

20

Type	Mnemonic	Function
Conditional Move	MOVF	Move Conditional on Floating Point False
	MOVN	Move Conditional on Not Zero
	MOVT	Move Conditional on Floating Point True
	MOVZ	Move Conditional on Zero

Type	Mnemonic	Function
Prefetch	PREF	Prefetch Register+Offset
NOP	NOP	No Operation

(e) Coprocessor Instructions

- The MIPS architecture defines four coprocessors (designated CP0, CP1, CP2, and CP3).
  - Coprocessor 0 (CP0) is incorporated on the CPU chip and supports the virtual memory system and exception handling. CP0 is also referred to as the System Control Coprocessor.
  - Coprocessor 1 (CP1) is reserved for the floating point coprocessor.
  - Coprocessor 2 (CP2) is available for specific implementations.
  - Coprocessor 3 (CP3) is available for future extensions.
- These instructions are not discussed here.

- MIPS32 architecture also supports a set of floating-point registers and floating-point instructions.
  - Shall be discussed later.

MIPS PROGRAMMING EXAMPLES

Some Examples of MIPS32 Arithmetic

C Code

```
A = B + C;
```

MIPS32 Code

```
add     $s1, $s2, $s3
```

B loaded in \$s2
C loaded in \$s3
A ← \$s1

C Code

```
A = B + C – D;
E = F + A;
```

MIPS32 Code

```
add     $t0, $s1, $s2
sub     $s0, $t0, $s3
add     $s4, $s5, $s0;
```

B loaded in \$s1
C loaded in \$s2
D loaded in \$s3
F loaded in \$s5
\$t0 is a temporary
A ← \$s0; E ← \$s4

25

Example on LOAD and STORE

C Code

```
A[10] = X – A[12];
```

MIPS32 Code

```
lw      $t0, 48($s3)
sub     $t0, $s2, $t0
sw      $t0, 40($s3);
```

\$s3 contains the starting address of the array A

\$s2 loaded with X

\$t0 is a temporary

Address of A[10] will be \$s3+40 (4 bytes per element)

Address of A[12] will be \$s3+48

26

Examples on Control Constructs

C Code

```
if (x==y)  z = x – y;
```

MIPS32 Code

```
        bne     $s0, $s1, Label
        sub     $s3, $s0, $s1
Label:  ......
```

$s0 loaded with x
$s1 loaded with y
z ← $s3

C Code

```
if (x != y)  z = x – y;
else         z = x + y;
```

MIPS32 Code

```
        beq     $s0, $s1, Lab1
        sub     $s3, $s0, $s1
        j       Lab2
Lab1:   add     $s3, $s0, $s1
Lab2:   ....
```

$s0 loaded with x
$s1 loaded with y
z ← $s3

- MIPS32 supports a limited set of conditional branch instructions:

```
beq   $s2,Label    // Branch to Label of $s2 = 0
bne   $s2,Label    // Branch to Label of $s2 != 0
```

- Suppose we need to implement a conditional branch after comparing two registers for less-than or greater than.

C Code

```
if (x < y)   z = x – y;
else         z = x + y;
```

MIPS32 Code

```
        slt    $t0,$s0,$s1
        beq    $t0, $zero, Lab1
        sub    $s3, $s0, $s1
        j      Lab2
Lab1:   add    $s3, $s0, $s1
Lab2:   ....
```

Set if less than.
If $s0 < $s1, then set $t0=1; else $t0=0.

- MIPS32 assemblers supports several pseudo-instructions that are meant for user convenience.
  - Internally the assembler converts them to valid MIPS32 instructions.
- Example: The pseudo-instruction branch if less than

blt $s1, $s2, Label

MIPS32 Code

```
        slt    $at, $s1, $s2
        bne    $t0, $zero, Label
        ....
Label:  ....
```

The assembler requires an extra register to do this.

The register $at (= R1) is reserved for this purpose.

Working with Immediate Values in Registers

- **Case 1**: Small constants, which can be specified in 16 bits.
  - Occurs most frequently (about 90% of the time).
  - Examples:

```
A = A + 16;    →   addi   $s1, $s1, 16      (A in $s1)
X = Y – 1025;  →   subi   $s1, $s2, 1025    (X in $s1, Y in $s2)
A = 100;       →   addi   $s1, $zero, 100   (A in $s1)
```

- **Case 2**: Large constants, that require 32 bits to represent.
  - How to load a large constant in a register?
  - Requires two instructions.
    - A "*Load Upper Immediate*" instruction, that loads a 16-bit number into the upper half of a register (lower bits filled with zeros).
    - An "*OR Immediate*" instruction, to insert the lower 16-bits.
  - Suppose we want to load 0xAAAA3333 into a register $s1.

lui $s1, 0xAAAA	1010101010101010	0000000000000000
ori $s1, $s1, 0x3333	1010101010101010	0011001100110011

Other MIPS Pseudo-instructions

Pseudo-Instruction	Translates to	Function
blt $1, $2, Label	slt $at, $1, $2 bne $at, $zero, Label	Branch if less than
bgt $1, $2, Label	sgt $at, $1, $2 bne $at, $zero, Label	Branch if greater than
ble $1, $2, Label	sle $at, $1, $2 bne $at, $zero, Label	Branch if less or equal
bge $1, $2, Label	sge $at, $1, $2 bne $at, $zero, Label	Branch if greater or equal
li $1, 0x23ABCD	lui $1, 0x0023 ori $1, $1, 0xABCD	Load immediate value into a register

Pseudo-Instruction	Translates to	Function
move $1, $2	add $1, $2, $zero	Move content of one register to another
la $a0, 0x2B09D5	lui $a0, 0x002B ori $a0, $a0, l0x09D5	Load address into a register

A Simple Function

C Function

```
swap (int A[], int k)
{
   int  temp;
   temp = A[k];
   A[k] = A[k+1];
   A[k+1] = temp;
}
```

MIPS32 Code

```
swap:  muli    $t0, $s0, 4
       add     $t0, $s1, $t0
       lw      $t1, 0($t0)
       lw      $t2, 4($t0)
       sw      $t2, 0($t0)
       sw      $t1, 4($t0)
       jr      $ra
```

$s0 loaded with index k

$s1 loaded with base address of A

Address of A[k] = $s1 + 4 * $s0

Exchange A[k] and A[k+1]

35

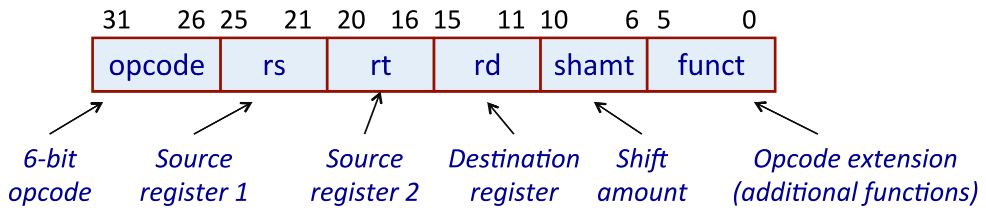
MIPS Instruction Encoding

- All MIPS32 instructions can be classified into three groups in terms of instruction encoding.
  - R-type (Register), I-type (Immediate), and J-type (Jump).
  - In an instruction encoding, the 32 bits of the instruction are divided into several fields of fixed widths.
  - All instructions may not use all the fields.
- Since the relative positions of some of the fields are same across instructions, instruction decoding becomes very simple.

36

(a) R-type Instruction Encoding

- Here an instruction can use up to three register operands.
  - Two source and one destination.
- In addition, for shift instructions, the number of bits to shift can also be specified.



- Examples of R-type instructions:

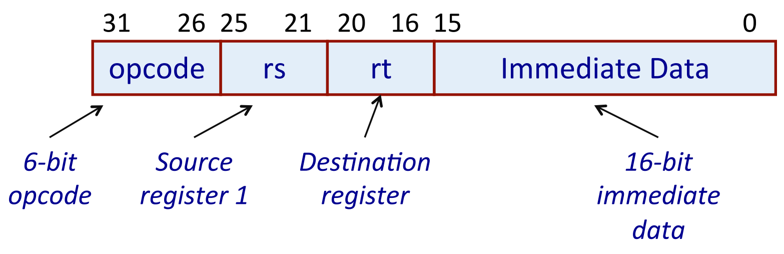
```
add     $s1, $s2, $s3
sub     $t1, $s3, $s4
sla     $s1, $s2, 5      // shift left $s2 by 5 places, and store in $s1
```

- An example instruction encoding: *add $t1, $s1, $s2*
  - Recall: $t1 is R9, $s1 is R17, and $s2 is R18.
  - For "add", opcode = 000000, and funct = 100000,

31 26	25 21	20 16	15 11	10 6	5 0
000000	10001	10010	01001	00000	100000

(b) I-type Instruction Encoding

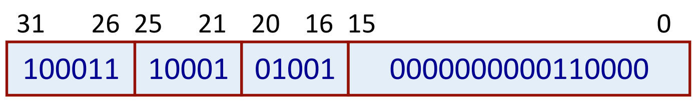
- Contains a 16-bit immediate data field.
- Supports one source and one destination register.



- Examples of I-type instructions:

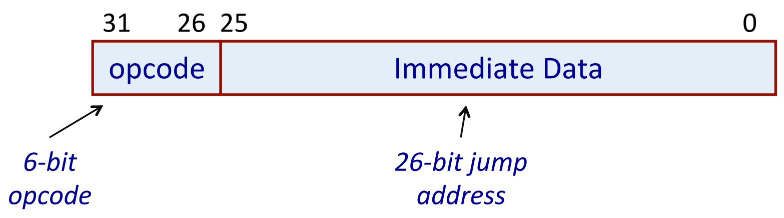
```
lw      $s1, 50($s5)
sw      $t1, 100($s1)
addi    $t0, $s1, 188
beq     $s1, $s2, Label      // Label is encoded as a 16-bit offset relative to PC
bne     $s3, $zero, Label
```

- An example instruction encoding: *lw $t1, 48($s1)*
  - Recall: $t1 is R9, $s1 is R17.
  - For "lw", opcode = 100011.

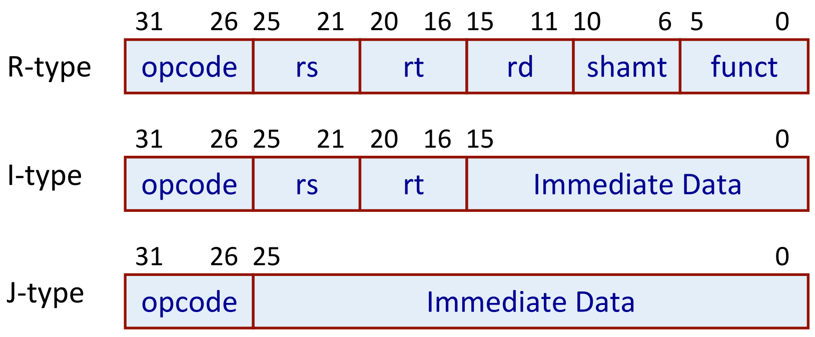


(c) J-type Instruction Encoding

- Contains a 26-bit jump address field.
  - Extended to 28 bits by padding two 0's on the right.
- Example: *j Label*



A Quick View



- Some instructions require two register operands *rs* & *rt* as input, while some require only *rs*.
- Gets known only after instruction is decoded.
- While decoding is going on, we can prefetch the registers in parallel.
  - May or may not be required later.

- Similarly, the 16-bit and 26-bit immediate data are retrieved and sign-extended to 32-bits in case they are required later.

Addressing Modes in MIPS32

• Register addressing	*add*	*$s1, $s2, $s3*
• Immediate addressing	*addi*	*$s1, $s2, 200*
• Base addressing	*lw*	*$s1, 150($s2)*

- Content of a register is added to a "base" value to get the operand address.

• PC relative addressing	*beq*	*$s1, $s2, Label*

- 16-bit offset is added to PC to get the target address.

• Pseudo-direct addressing	*j*	*Label*

- 26-bit offset if shifted left by 2 bits and then added to PC to get the target address.

43

AMDAHL'S LAW

44

Introduction

Gene Amdahl

- Amdahl's law was established in 1967 by Gene Amdahl.
- Basically provides an understanding on scaling, limitations and economics of parallel computing.
- Forms the basis for quantitative principles in computer system design.
  - Can be applied to other application domains as well.

45

What is Amdahl's Law?

- It can be used to find the maximum expected improvement of an overall system when only *part of the system* is improved.
- It basically states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Very useful to check whether any proposed improvement can provide expected return.
  - Used by computer designers to enhance only those architectural features that result in reasonable performance improvement.
  - Referred to as *quantitative principles in design*.

46

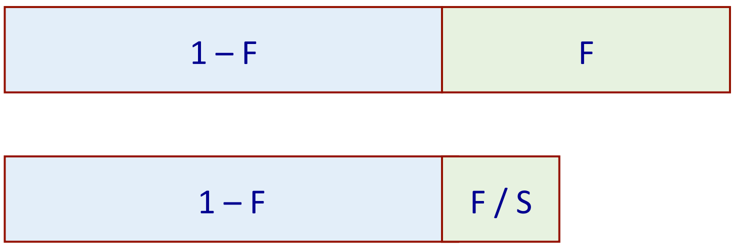
- Amdahl's law demonstrates the *law of diminishing returns*.
- An example:
  - Suppose we are improving a part of the computer system that affects only 25% of the overall task.
  - The improvement can be *very little* or *extremely large*.
  - With "*infinite*" speedup, the 25% of the task can be done in "*zero*" time.
  - Maximum possible speedup = XT_{orig} / XT_{new} = $1 / (1 – 0.25)$ = 1.33

We can never get a speedup of more than 1.33

- Amdahl's law concerns the speedup achievable from an improvement in computation that affects a fraction *F* of the computation, where the improvement has a speedup of *S*.

Before improvement

After improvement



- Execution time before improvement: *(1 – F) + F = 1*
- Execution time after improvement: *(1 – F) + F / S*
- Speedup obtained:

$$\boxed{\text{Speedup} = \frac{1}{(1 - F) + F / S}}$$

- As *S → ∞*, *Speedup → 1 / (1 – F)*
  - The fraction *F* limits the maximum speedup that can be obtained.

- Illustration of law of diminishing returns:
  - Let *F = 0.25*.
  - The table shows the speedup (*= 1 / (1 – F + F / S*) for various values of *S*.

1 / (1 – 0.25) = 1.33

S	Speedup
1	1.00
2	1.14
5	1.25
10	1.29

S	Speedup
50	1.32
100	1.33
1000	1.33
100,000	1.33

- Illustration of law of diminishing returns:
  - Let *F = 0.75*.
  - The table shows the speedup for various values of *S*.

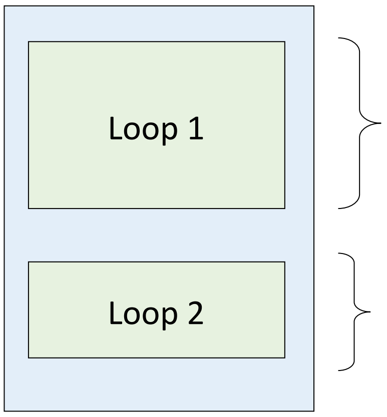
1 / (1 – 0.75) = 4.00

S	Speedup
1	1.00
2	1.60
5	2.50
10	3.08

S	Speedup
50	3.77
100	3.88
1000	3.99
100,000	4.00

51

Design Alternative using Amdahl's law



Loop 1	500 lines	10% of total execution time
Loop 2	20 lines	90% of total execution time

52

- Some examples:
  - We make 10% of a program 90X faster, speedup = 1 / (0.9 + 0.1 / 90) = 1.11
  - We make 90% of a program 10X faster, speedup = 1 / (0.1 + 0.9 / 10) = 5.26
  - We make 25% of a program 25X faster, speedup = 1 / (0.75 + 0.25 / 25) = 1.32
  - We make 50% of a program 20X faster, speedup = 1 / (0.5 + 0.5 / 20) = 1.90
  - We make 90% of a program 50X faster, speedup = 1 / (0.1 + 0.9 / 50) = 8.47

53

Example 1

- Suppose we are running a set of programs on a RISC processor, for which the following instruction mix is observed:

Operation	Frequency	CPI_i
Load	20 %	5
Store	8 %	3
ALU	60 %	1
Branch	12 %	2

$W_i * CPI_i$	% Time
1.00	0.48
0.24	0.12
0.60	0.29
0.24	0.11

CPI = 2.08

$\rightarrow$ 1 / 2.08

We carry out a design enhancement by which the CPI of Load instructions reduces from 5 to 2. What will be the overall performance improvement?

54

Fraction enhanced F = 0.48

Fraction unaffected 1 – F = 1 – 0.48 = 0.52

Enhancement factor S = 5 / 2 = 2.5

Therefore, speedup is

$$\frac{1}{(1 - F) + F / S} = \frac{1}{0.52 + 0.48 / 2.5} = 1.40$$

55

Example 2

- The execution time of a program on a machine is found to be 50 seconds, out of which 42 seconds is consumed by multiply operations. It is required to make the program run 5 times faster. By how much must the speed of the multiplier be improved?
  - Here, F = 42 / 50 = 0.84
  - According to Amdahl's law,

$$5 = 1 / (0.16 + 0.84 / S)$$

or, $0.80 + 4.2 / S = 1$

or, $S = 21$

56

Example 2a

- The execution time of a program on a machine is found to be 50 seconds, out of which 42 seconds is consumed by multiply operations. It is required to make the program run **8** times faster. By how much must the speed of the multiplier be improved?
  - Here, F = 42 / 50 = 0.84
  - According to Amdahl's law,

$$8 = 1 / (0.16 + 0.84 / S)$$

or, $1.28 + 6.72 / S = 1$

or, $S = -24$

No amount to speed improvement in the multiplier can achieve this.

Maximum speedup achievable:
$1 / (1 - F) = 6.25$

57

Example 3

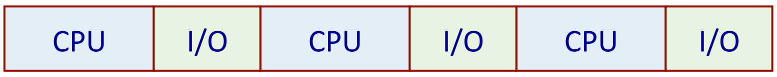
- Suppose we plan to upgrade the processor of a web server. The CPU is 30 times faster on search queries than the old processor. The old processor is busy with search queries 80% of the time. Estimate the speedup obtained by the upgrade.
  - Here, F = 0.80 and S = 30
  - Thus, speedup = $1 / (0.20 + 0.80 / 30) = 4.41$

58

Example 4

- The total execution time of a typical program is made up of 60% of CPU time and 40% of I/O time. Which of the following alternatives is better?
  a) Increase the CPU speed by 50%
  b) Reduce the I/O time by half

  Assume that there is no overlap between CPU and I/O operations.



59

- Increase CPU speed by 50%
  - Here, F = 0.60 and S = 1.5
  - Speedup = 1 / (0.40 + 0.60 / 1.5) = 1.25
- Reduce the I/O time by half
  - Here, F = 0.40 and S = 2
  - Speedup = 1 / (0.60 + 0.40 / 2) = 1.25

Thus, both the alternatives result in the same speedup.

60

Example 5

- Suppose that a compute-intensive bioinformatics program is running on a given machine X, which takes 10 days to run. The program spends 25% of its time doing integer instructions, and 40% of time doing I/O. Which of the following two alternatives provides a better tradeoff?
  - a) Use an optimizing compiler that reduces the number of integer instructions by 30% (assume all integer instructions take the same time).
  - b) Optimizing the I/O subsystem that reduces the latency of I/O operations from 10 µsec to 5 µsec (that is, speedup of 2).

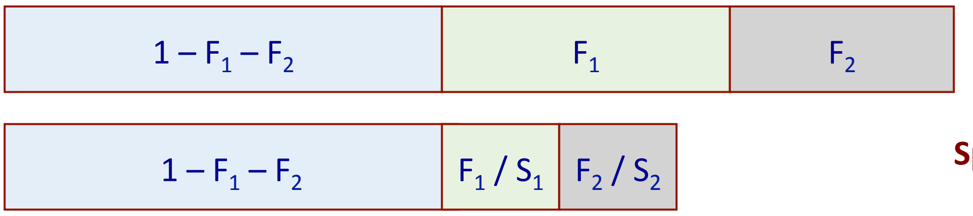
61

- Alternative (a):
  - Here, F = 0.25 and S = 100 / 70
  - Speedup = 1 / (0.75 + 0.25 * 70 / 100) = 1.08
- Alternative (b):
  - Here, F = 0.40 and S = 2
  - Speedup = 1 / (0.60 + 0.40 / 2) = 1.25

62

Extension to Multiple Enhancements

- Suppose we carry out multiple optimizations to a program:
  - Optimization 1 speeds up a fraction *F_1* of the program by a factor *S_1*
  - Optimization 2 speeds up a fraction *F_2* of the program by a factor *S_2*

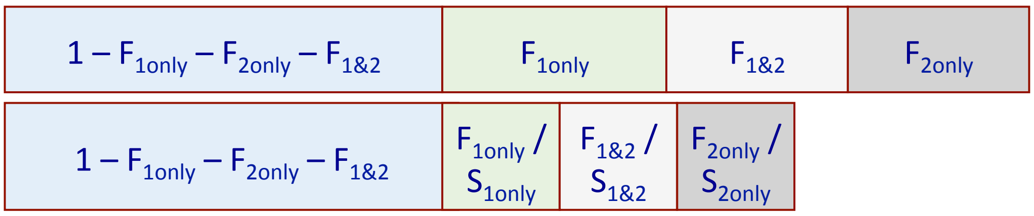


Speedup

$$\frac{1}{(1 - F_1 - F_2) + F_1 / S_1 + F_2 / S_2}$$

63

- In the calculation as shown, it is assumed that *F_1* and *F_2* are disjoint.
  - *S_1* and *S_2* do not apply to the same portion of execution.
- If it is not so, we have to treat the overlap as a separate portion of execution and measure its speedup independently.
  - F_{1only} , F_{2only} , and $F_{1\&2}$ with speedups S_{1only} , S_{2only} , and $S_{1\&2}$



$$\textbf{Speedup} = \frac{1}{(1 - F_{1only} - F_{2only} - F_{1\&2}) + F_{1only} / S_{1only} + F_{2only} / S_{2only} + F_{1\&2} / S_{1\&2}}$$

64

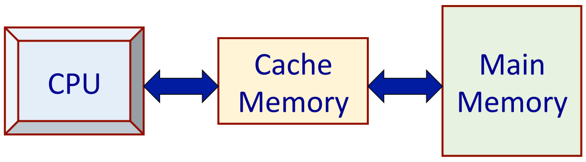
- General expression:
  - Assume *m* enhancements of fractions $F_1, F_2, \ldots, F_m$ by factors of $S_1, S_2, \ldots, S_m$ respectively.

$$\text{Speedup} = \frac{1}{(1 - \sum_{i=1}^{m} F_i) + \sum_{i=1}^{m} \frac{F_i}{S_i}}$$

65

Example 6

- Consider an example of memory system.
  - Main memory and a fast memory called cache memory.
  - Frequently used parts of program/data are kept in cache memory.
  - Use of the cache memory speeds up memory accesses by a factor of 8.
  - Without the cache, memory operations consume a fraction 0.40 of the total execution time.
  - Estimate the speedup.



Solution

$$\textbf{Speedup} = \frac{1}{(1-F) + F/S} = \frac{1}{(1-0.4) + 0.4/8} = 1.54$$

66

Example 7

- Now we consider two levels of cache memory, L1-cache and L2-cache.

  Assumptions:
  - Without the cache, memory operations take 30% of execution time.
  - The L1-cache speeds up 80% of memory operations by a factor of 4.
  - The L2-cache speeds up 50% of the remaining 20% memory operations by a factor of 2.

  We want to find out the overall speedup.

67

- Solution:
  - Memory operations = 0.3
  - $F_{L1} = 0.3 * 0.8 = 0.24$
  - $S_{L1} = 4$
  - $F_{L2} = 0.3 * (1 – 0.8) * 0.5 = 0.03$
  - $S_{L2} = 2$

Speedup

$$\frac{1}{(1 - F_{L1} - F_{L2}) + F_{L1} / S_{L1} + F_{L2} / S_{L2}}$$

$$\frac{1}{(1 - 0.24 - 0.03) + 0.24 / 4 + 0.03 / 2}$$

= 1.24

68