CMPE 200

COMPUTER ARCHITECTURE

Lecture 2 – Instructions: Language of the Computer

Bo Yu
Computer Engineering Department
SJSU

Adapted from Computer Organization and Design, 5th Edition, 4th edition, Patterson and Hennessy, MK
and Computer Architecture – A Quantitative Approach, 4th edition, Patterson and Hennessy, MK

Lecture 1 Key Concepts Review

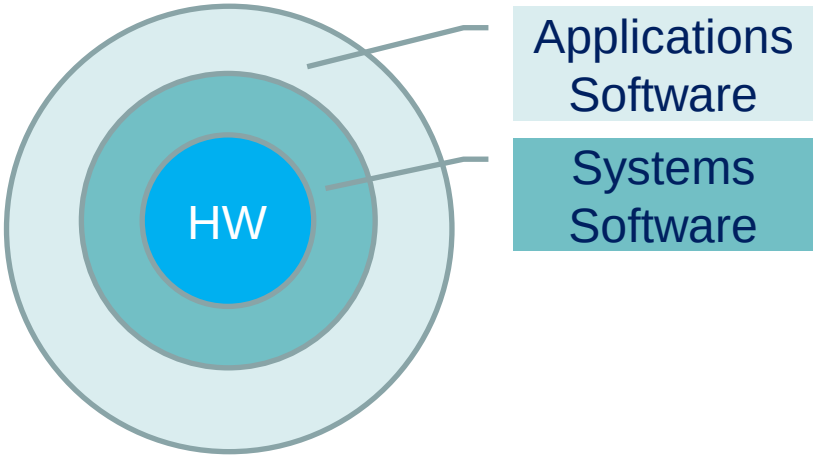
- Last Lecture:
  - Technology trends: Culture of tracking, anticipating and exploiting advances in technology
    - o Design techniques
    - o Machine structures
    - o Technology factors
    - o Evaluation methods
  - Quantitative Comparisons
    - o Power
    - o Performance
    - o Cost

Lecture 1 Key Concepts Review

- CH 1:
  - Classes of Computing Applications
    - o Personal Computer (PC)
    - o Server
    - o Supercomputer
    - o Embedded computer
    - o Personal Mobile Device (PMD)
    - o Cloud Computing
    - o Software as a Service (SaaS)
  - Hardware/Software Concentric circles
    - o Hardware
    - o Systems Software
      - ❖ Operating System (OS): Windows, Linux, IOS
        - ▪ Handling basic I/O Operations
        - ▪ Allocating storage and memory
        - ▪ Managing the HW resources for the benefit of multiple applications
      - ❖ Compiler: Translating high-level language into assembly language
      - ❖ Assembler: Translating assembly language into binary instructions
    - o Applications Software

- Common Size Term: Decimal vs. Binary

Decimal Term	Value	Binary Term	Value	% Larger
Kilobyte (KB)	10^3	Kibibyte (KiB)	2^{10}	2%
Megabyte (MB)	10^6	Mebibyte (MiB)	2^{20}	5%
Gigabyte (GB)	10^9	Gibibyte (GiB)	2^{30}	7%
Terabyte (TB)	10^{12}	Tebibyte (TiB)	2^{40}	10%



Lecture 1 Key Concepts Review

- CH 1:
  - Five Classic Components of the Standard Organization of a Computer
    - o Input
    - o Output
    - o Memory
    - o Datapath
    - o Control

    } Processor
  - Computer Network – Backbone of computer
    - o Communication, resource sharing, remote access
    - o LAN, WAN
  - Memory Hierarchy
    - o Main Memory: Volatile
    - o Secondary Memory: Non Volatile
    - o Memory speed, capacity, cost
  - Technology for Building Processors and Memory
    - o Vacuum tube -> Transistor -> Integrated Circuit -> VLSI -> ULSI
    - o Silicon -> Wafers -> Dies -> Packaged Dies
  - Power
    - o Static Power

      $$Power_{static} = Current_{static} \times Voltage$$
    - o Dynamic Power

      $$Power_{dynamic} = 1/2 \times CapacitiveLoad \times Voltage^2 \times FrequencySwitched$$
  - Performance
    - o Defining Performance
      - ❖ Response Time (Execution Time)
      - ❖ Throughput (Bandwidth)
    - o Measuring Performance
      - ❖ Execution Time
      - ❖ Clock Cycles, Clock Period
      - ❖ CPI (Clock cycles Per Instruction)
      - ❖ Execution Time=Instruction Count x CPI x Clock Period
      - ❖ MIPS=Instruction Count/(Execution Timex10^6)
    - o How HW/SW affect the three factors in the CPU performance equation?
      - ❖ Algorithm
      - ❖ Programming language
      - ❖ Compiler
      - ❖ ISA
      - ❖ Technology
    - o Amdahl's Law

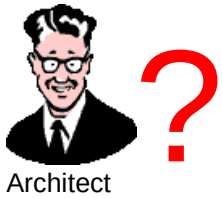
Lecture 2: Instructions: Language of the Computer

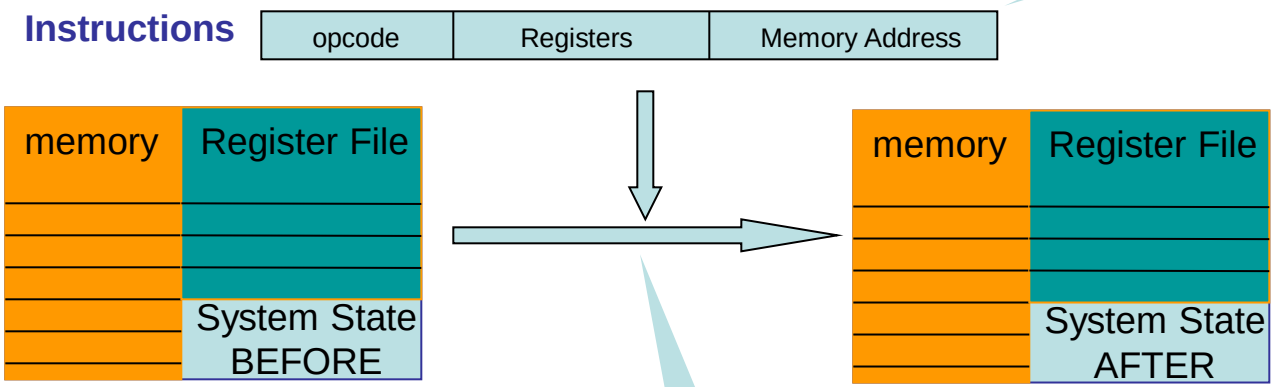
- Technology determines computer implementation:
  - Transistor size, number of transistors
  - VLSI
  - Memory technologies
  - Power
  - Packaging
- BUT Software is what measures performance of a computer
  - Programs running on computers have a huge impact on the architecture of the computers
  - HENCE, Instruction Set Architecture (ISA)
- What is “Computer Architecture”?
  - Computer Architecture = Instruction Set Architecture + Machine Organization

Lecture 2: Instructions: Language of the Computer

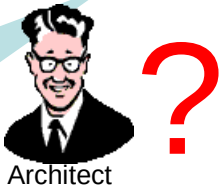
- ISA is the connection between the HW and SW:
  - ISA defines the state of the system and makes it visible to SW at any time
    - o HW defines the correct way to execute programs
    - o SW defines how programs will execute
  - ISA defines the instructions that control the state transitions of the system
    - o Sequence of steps to complete operations
  - **So, an ISA is composed of** :

Instruction Formats
Instruction Types
Addressing Modes

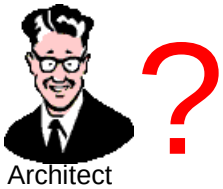




Machine State
Register File HW
Memory HW

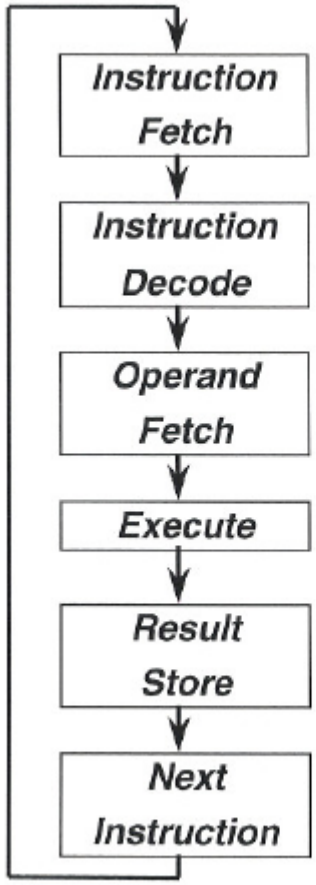


Opcode
Data
Events



Register file is a small set of high-speed storage cells inside CPU

Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor instruction

Adapted from Lecture Notes: Computer Organization & Design: The Hardware/Software Interface, David A. Patterson and John L. Hennessy.
Hal Katircioglu SJSU Cmpe 140 Spring 1999

- Processor State:
  - The information in the processor after an instruction is executed (or power up/reset or an event)
    - o Only a part of the system state is available (visible to software)
    - o Hardware has, by design, instructions that provide access to system state (though very limited) under specified privilege levels
      - √ Operating system vs applications

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Lecture 2: Instructions: Language of the Computer

- Different Computer Architectures:
  - **RISC: Reduced Instruction Set Computer -> ARM**
  - **CISC: Complex Instruction Set Computer -> x86**
- MIPS:
  - **Used as the example throughout the book**
  - **Microprocessor without Interlocked Pipelined Stages**
  - **Designed since 1980s**
  - **MIPS is a RISC instruction set architecture (ISA)**
  - **Developed by MIPS Computer Systems, now MIPS Technologies**
  - **Current version is MIPS32/64 Release 6**
  - **MIPS architecture greatly influenced later RISC architectures**
  - **Similar to other architectures**
  - **Large share of embedded core market**
    - **Applications in consumer electronics, network/storage equipment, cameras, printers, …**
- Three Other Popular Instruction Sets:
  - **ARMv7 (32bits)**
  - **ARMv8 (64bits)**
  - **Intel x86**

Instructions:

- A set of possible operations
  - o More primitive than higher level languages
  - o Number of operations
  - o should be easy to decode
  - o Register type, memory-type, control-type, etc,…
  - o Very restrictive (i.e. MIPS Arithmetic Instructions)

- Should ideally address ALL resources in system
  - o Sequence of steps to complete operations

- Instruction length :
  - o Variable length more difficult to decode – **RISC vs CISC**

Lecture 2: Instructions: Language of the Computer

- Operations on Data:
  - Data move instruction:
    - o Register
    - o Stack
    - o Memory
    - o IO
  - Arithmetic and Logic
    - o Compare
    - o ADD/SUB
    - o MUL/DIV
    - o …..
  - Shift and Rotate
  - Bit Manipulation
    - o Clear
    - o Set
    - o Invert
  - ……..

- Instruction Decoding:
  - 3-address instruction format:
    - o Opcode – Dest – Src1 – Src2
    - o Used by register-register architectures
  - 2-address instruction format
    - o Opcode – Dest/Src1 – Src2
    - o Used by register-memory architectures
  - 1-address instruction format:
    - o Opcode – Src
    - o Used by accumulator architectures
  - 0-address instruction format
    - o Opcode
    - o Used by stack architectures

- Various Instruction Format examples:

Operation and no. of operands	Address specifier 1	Address field 1	• • •	Address specifier *n*	Address field *n*

(a) Variable (e.g., Intel 80x86, VAX)

Operation	Address field 1	Address field 2	Address field 3

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

MIPS Operands and Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words, all instructions a single size
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity! (Design Principle 1: Simplicity favors regularity)
- MIPS 32 General Purpose Registers (fast locations for data)
  - Design Principle 2: Smaller is faster

Register Number	Conventional Name	Usage
$0	$zero	Hard-wired 0
$1	$at	Reserved by assembler to handle large constants
$2-$3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions, not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary data, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved by kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

Lecture 2: Instructions: Language of the Computer

MIPS Instructions and Assembly Language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

MIPS Instruction Set Formats

- Design Principle 3: Good design demands good compromises
- Keep all instructions the same length, different formats for different instructions
- R-type (Register), I-type (Immediate), J-type (Jump)

Basic instruction formats

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15		
J	opcode	address				
	31 26	25				

Floating-point instruction formats

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15		

Adapted from Prof. David Patterson

Lecture 2: Instructions: Language of the Computer

Arithmetic Operation:

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a ← b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register Operands:

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - cf. main memory: millions of locations

- C code:

```
f = (g + h) - (i + j);
```

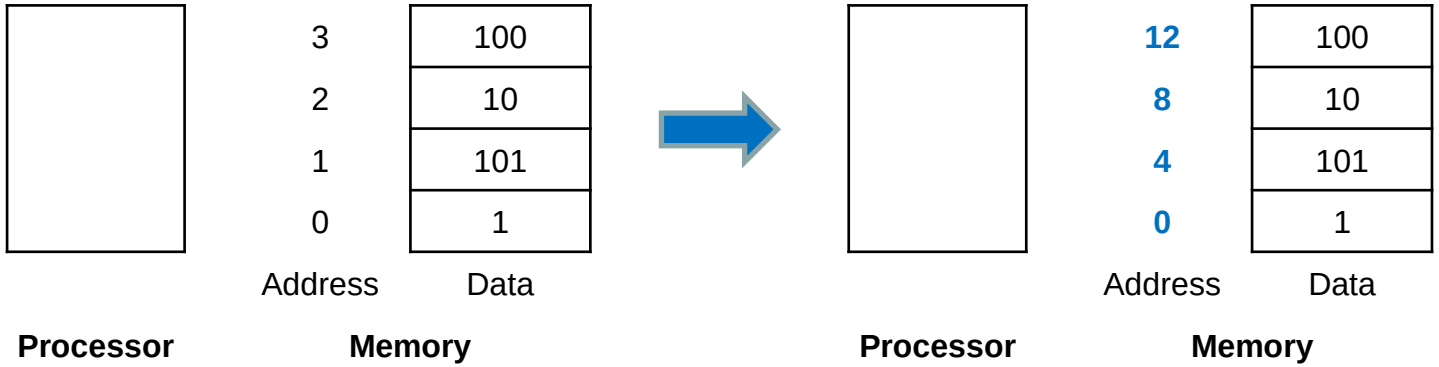
  – f, …, j in $s0, …, $s4

- Compiled MIPS code:

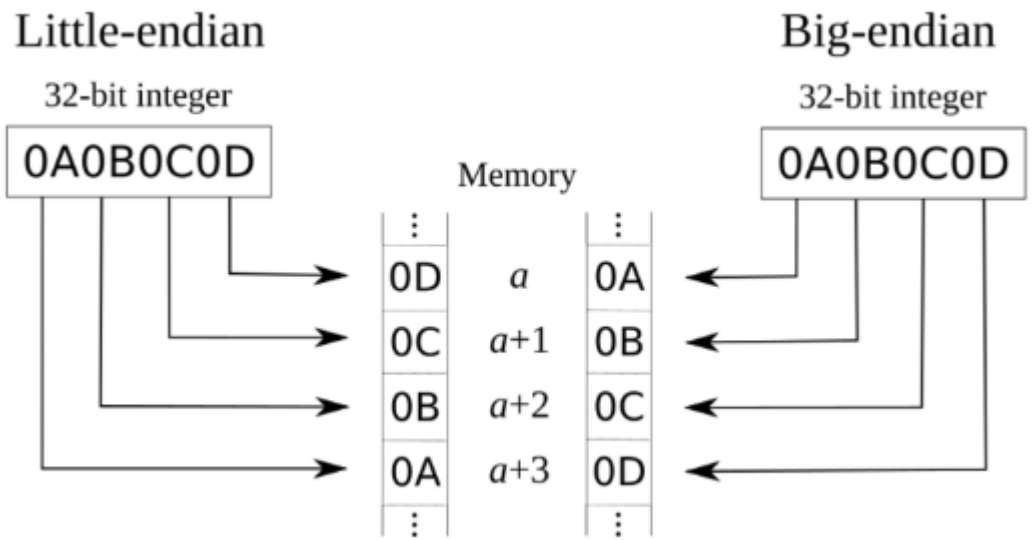
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

- Memory Operands:
  - Main memory used for composite data
    - Arrays, structures, dynamic data
  - To apply arithmetic operations
    - Load values from memory into registers
    - Store result from register to memory
  - Memory is byte addressed
    - Each address identifies an 8-bit byte
  - Words are aligned in memory
    - Address must be a multiple of 4
  - MIPS is Big Endian
    - Most-significant byte at least address of a word
    - *c.f.* Little Endian: least-significant byte at least address

Word address must be a multiple of 4



Big Endian vs. Little Endian



Memory Operand Example 1:

- C code:

```
g = h + A[8];
```

  – g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:
  – Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```

offset

base register

- Memory Operand Example 2:
  - C code:

```
A[12] = h + A[8];
```

    - h in $s2, base address of A in $s3
  - Compiled MIPS code:
    - Index 8 requires offset of 32

```
lw  $t0, 32($s3)     # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

- Immediate Operands:
  - Constant data specified in an instruction

```
addi $s3, $s3, 4
```

  - No subtract immediate instruction
    - Just use a negative constant

```
addi $s2, $s1, -1
```

  - *Design Principle 3*: Good design demands good compromises
    - Small constants are common
    - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Polling: MIPS to C

- Assume variables f, g, h, i are assigned to registers $s0-$s3 respectively. For the following MIPS assembly instructions, what is the corresponding C code?

```
add $s0, $s2, $s1
sub $s1, $s3, $s0
```

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

Binary signed and unsigned number

- Binary to Decimal Conversion
  - In any number base the value of ith digit d is: **d x Base^i**
  - $1011_{two} = ((1x2^3)+(0x2^2)+(1x2^1)+(1x2^0))_{ten} = 11_{ten}$
  - MIPS 32bits Word: $(d31x2^{31})+(d30x2^{30})+(d29x2^{29})+\ldots+(d1x2^1)+(d0x2^0)$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

- Least Significant Bit (rightmost,bit0), Most Significant Bit (leftmost,bit31)
- Unsigned numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

……..

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Binary signed and unsigned number

- Representation of Signed numbers
  - Sign and magnitude: Add a separate sign represented in a single bit ✗
  - Two's Complement Representation (every computer uses today)
    - leading 0s mean positive, leading 1s mean negative
  - 32bits (MSB-sign bit): $(d31x\text{-}2^{31})+(d30x2^{30})+(d29x2^{29})+\ldots+(d1x2^{1})+(d0x2^{0})$
- Signed numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
……..
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten} = (2^{31}\text{-}3)$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten} = (2^{31}\text{-}2)$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten} = (2^{31}\text{-}1)$
$\mathbf{1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten} = (-2^{31})}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten} = (-2^{31}+1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten} = (-2^{31}+2)$
……..
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

2s-Complement Signed Binary Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Binary Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

- Signed Negation
  - Complement and add 1
    - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

  - Example: negate +2
    - +2 = 0000 0000 … 0010_2
    - –2 = 1111 1111 … 1101_2 + 1
      = 1111 1111 … 1110_2

Negation of Binary signed and unsigned number

- Invert every 0 to 1 and every 1 to 1 of X, the sum of X and XI (inverted) is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

  X + XI (inverted) = -1 → XI (inverted) + 1 = -X

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_{two} = 4_{ten}$

  Negating the number by inverting the bits and adding one:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{two}$

  $+ \qquad 1_{two}$

  $=1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two} = -4_{ten}$

- Sign Extension: Convert 16-bit 2_{ten} and -2_{ten} to 32-bit binary numbers

  16-bit: $0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

  32-bit: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

  Negate the 16-bit version of 2: $1111\ 1111\ 1110_{two} = -2_{ten}$

  Copy the sign bit 16 times to the left:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

Signed Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Polling: Two's Complement Number

- What is the decimal value of this 32-bit two's complement number?

1111 1111 1111 1111 1111 1111 1111 1000

a) -4_{ten}

b) -8_{ten}

c) -16_{ten}

d) -32_{ten}

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

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 (00000 for now)
  - funct: function code (extends opcode)

R-Format 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

$$00000010001100100100000000100000_2 = 02324020_{16}$$

- Hexadecimal
  - Base 16
    - Compact representation of bit strings
    - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

  - Example: eca8 6420
    - 1110 1100 1010 1000 0110 0100 0010 0000

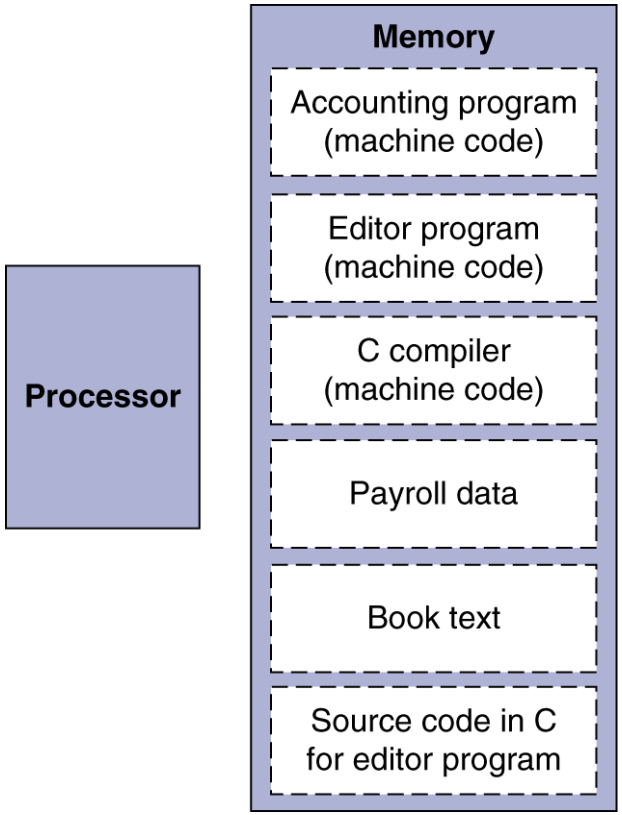
I-Format Example

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
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

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)

Polling: MIPS to C

- Assume variables f, g, h, i are assigned to registers $s0-$s3 respectively. Assume the base address of array A is in register $s4. Assume arrays A is 4-byte word. For the following MIPS assembly instructions, what is the corresponding C code?

```
sll $t0, $s2, 2
add $t1, $t0, $s4
lw $t0, 0($t1)
add $s1, $t0, $s0
```

Lecture 2 Key Concepts Review

- 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

Lecture 2 Key Concepts Review

- 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

Lecture 2 Key Concepts Review

- 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)

Register 0: always read as zero

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

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

Lecture 2 Key Concepts Review

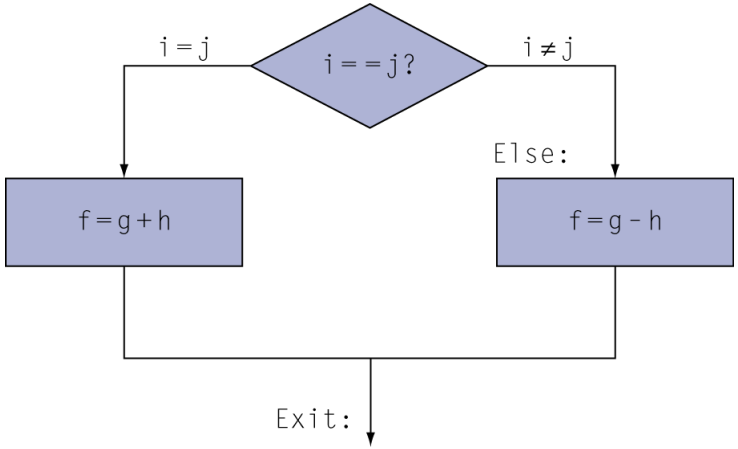
- Conditional Operations
  - Branch to a labeled instruction if a condition is true
    - Otherwise, continue sequentially
  - beq rs, rt, L1
    - if (rs == rt) branch to instruction labeled L1;
  - bne rs, rt, L1
    - if (rs != rt) branch to instruction labeled L1;
  - j L1
    - unconditional jump to instruction labeled L1

Lecture 2 Key Concepts Review

- Compiling If Statements
  - C code:

```
if (i==j) f = g+h;
else f = g-h;
```

    - f, g, … in $s0, $s1, …



  - Compiled MIPS code:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   …
```

Assembler calculates addresses

Lecture 2 Key Concepts Review

- Compiling Loop Statements
  - C code:

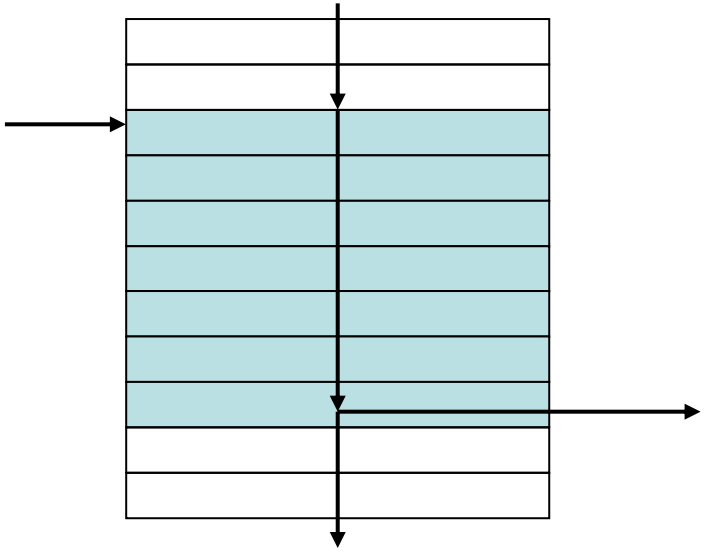
```
while (save[i] == k) i += 1;
```

  – i in $s3, k in $s5, address of save in $s6

  - Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

- Basic Blocks
  - A basic block is a sequence of instructions with
    - No embedded branches (except at end)
    - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- slt rd, rs, rt
  - if (rs < rt) rd = 1; else rd = 0;
- slti rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Lecture 2 Key Concepts Review

- Signed vs. Unsigned Comparison
  - Signed comparison: slt, slti
  - Unsigned comparison: sltu, sltui
  - Example
    - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
    - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
    - slt $t0, $s0, $s1 # signed
      - $-1 < +1 \Rightarrow$ $t0 = 1
    - sltu $t0, $s0, $s1 # unsigned
      - $+4,294,967,295 > +1 \Rightarrow$ $t0 = 0

Lecture 2 Key Concepts Review

- Procedure Calling
  - Steps required
    1. Place parameters in registers
    2. Transfer control to procedure
    3. Acquire storage for procedure
    4. Perform procedure's operations
    5. Place result in register for caller
    6. Return to place of call

Lecture 2 Key Concepts Review

- Register Usage
  - $a0 – $a3: arguments (reg's 4 – 7)
  - $v0, $v1: result values (reg's 2 and 3)
  - $t0 – $t9: temporaries
    - Can be overwritten by callee
  - $s0 – $s7: saved
    - Must be saved/restored by callee
  - $gp: global pointer for static data (reg 28)
  - $sp: stack pointer (reg 29)
  - $fp: frame pointer (reg 30)
  - $ra: return address (reg 31)

Lecture 2 Key Concepts Review

- Procedure Call Instructions
  - Procedure call: jump and link

    ```
    jal ProcedureLabel
    ```

    - Address of following instruction put in $ra
    - Jumps to target address

  - Procedure return: jump register

    ```
    jr $ra
    ```

    - Copies $ra to program counter
    - Can also be used for computed jumps
      - e.g., for case/switch statements

Lecture 2 Key Concepts Review

- Leaf Procedure Example (no subroutine calls)
  - C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

- Leaf Procedure Example
  - MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Lecture 2 Key Concepts Review

- Non-Leaf Procedures
  - Procedures that call other procedures
  - For nested call, caller needs to save on the stack:
    - Its return address
    - Any arguments and temporaries needed after the call
  - Restore from the stack after the call

Lecture 2 Key Concepts Review

- Non-Leaf Procedure Example
  - C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

  - Argument n in $a0
  - Result in $v0

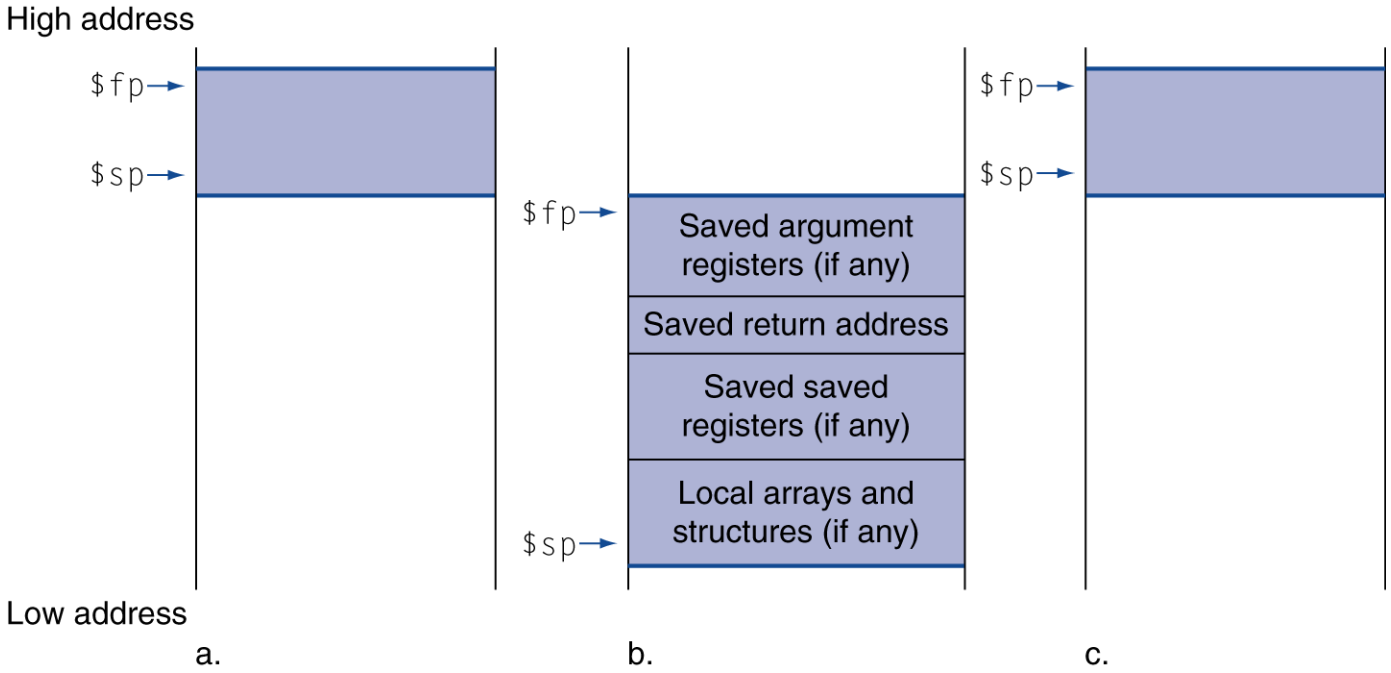
Lecture 2 Key Concepts Review

- Non-Leaf Procedure Example
  - MIPS code:

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

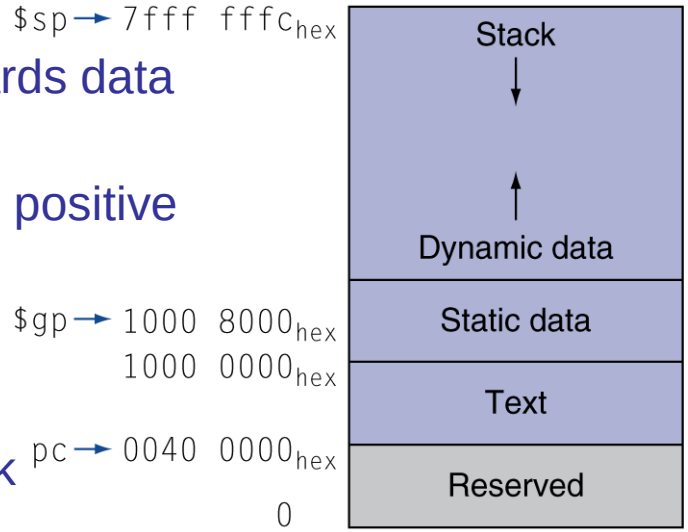
Lecture 2 Key Concepts Review

- Local Data on the Stack
  - Local data allocated by callee
    - e.g., C automatic variables
  - Procedure frame (activation record)
    - Used by some compilers to manage stack storage



Lecture 2 Key Concepts Review

- Memory Layout
  - Text: program code
  - Static data: global variables
    - e.g., static variables in C, constant arrays and strings
    - $gp initialized to address allowing ±offsets into this segment
    - $sp → starts at 7fff fffc, grows down towards data segment
    - $gp → starts at 1000 8000, moves using positive and negative 16-bit offset
    - Text Segment: MIPS machine code
    - Static data: starts at 1000 0000
    - Dynamic data: heap grows towards stack
  - Dynamic data: heap
    - E.g., malloc in C, new in Java
  - Stack: automatic storage



Lecture 2 Key Concepts Review

- MIPS ISA Three Special Purpose Registers (SPR)
  - PC (Program Counter)
    - A special register containing the address of instruction in the program being executed. Instruction fetching occurs at the address in PC.
    - Not directly visible and manipulated by programmers in MIPS
  - HI/LO Registers
    - Two 32-bit registers (HI and LO) that hold results of integer multiply and divide

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lui $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

- Branch Addressing
  - Branch instructions specify
    - Opcode, two registers, target address
  - Most branch targets are near branch
    - Forward or backward

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

- Jump Addressing
  - Jump (j and jal) targets could be anywhere in text segment
    - Encode full address in instruction

op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31...28}$: (address × 4)

- Target Addressing Example
  - Loop code from earlier example (on page 49)
    - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

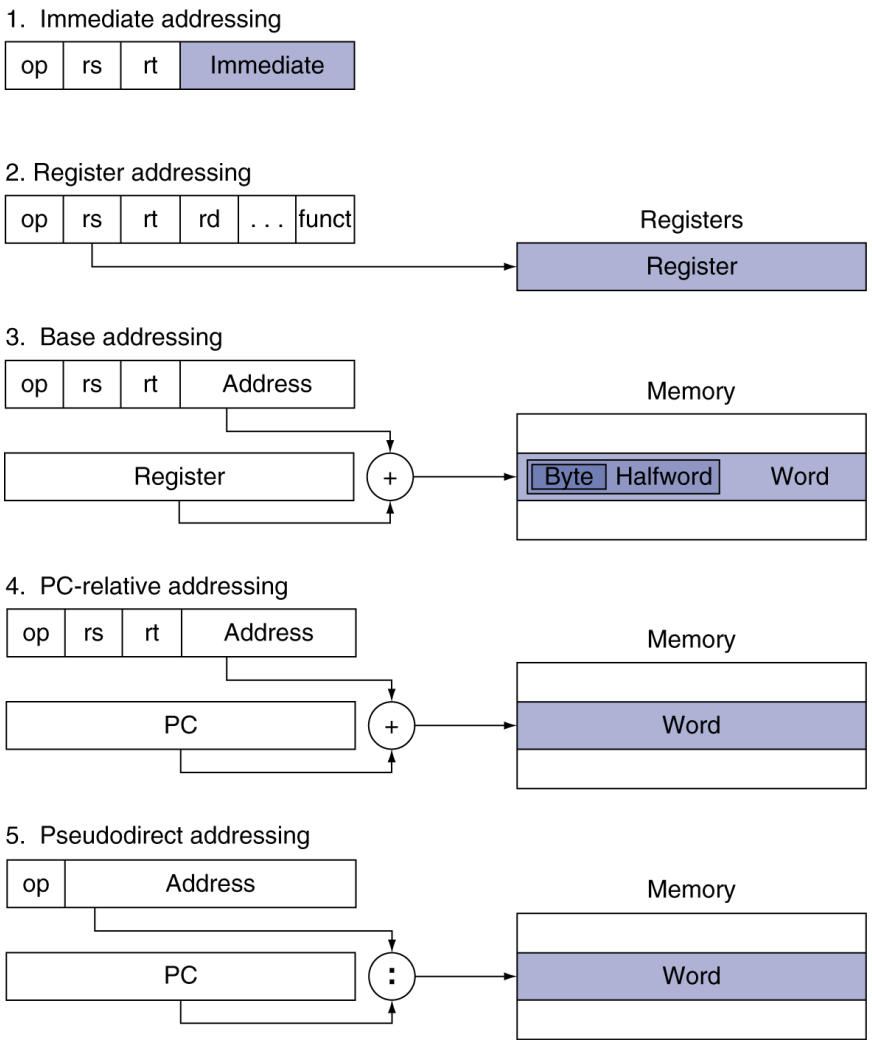
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
                ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

Lecture 2 Key Concepts Review

- Addressing Mode Summary



Lecture 2: Instructions: Language of the Computer

- Data Access – Addressing Modes:
  - Immediate Mode
    - o Operand is part of the instruction
  - Register Mode:
    - o Operand is in the register
  - Base or Displacement Mode
    - o The effective address of the operand is the sum of the contents of a register and a value – displacement – given in the instruction
  - PC-relative Mode
    - o A displacement is added to the PC
  - Pseudodirect Mode
    - o The jump address is the 26 bits of the instruction concatenated with 4 upper bits of the PC
  - Direct Mode
    - o The address of the operand in memory is in the instruction
  - Register Indirect Mode
    - o The address of the operand in memory is in one of the registers
  - Auto-increment/Auto-decrement
    - o Same as register indirect, except that the contents of the register is incremented/decremented after the use of the address – good for loops
  - Indexed and scaled-indexed Mode
    - o Similar to register-indirect - the address of the operand is in a register called the index register that may be scaled by a factor (e.g., 1, 2, 4, 8, 16).
  - Indirect scaled indexed Mode
  - Indirect scaled indexed with displacement Mode

Lecture 2: Instructions: Language of the Computer

- Data Storage – Address Space:
  - Separate Register Files for
    - o Integer ops
    - o FP ops
    - o Multimedia
    - o …..

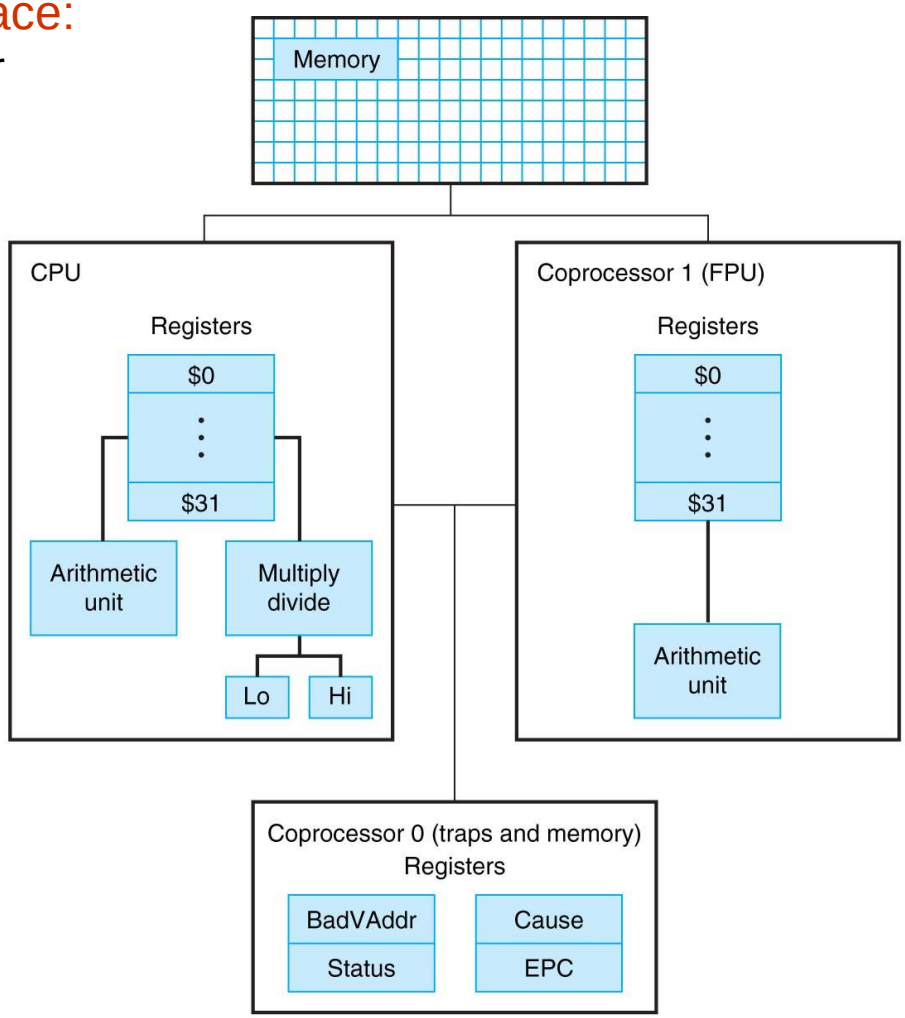
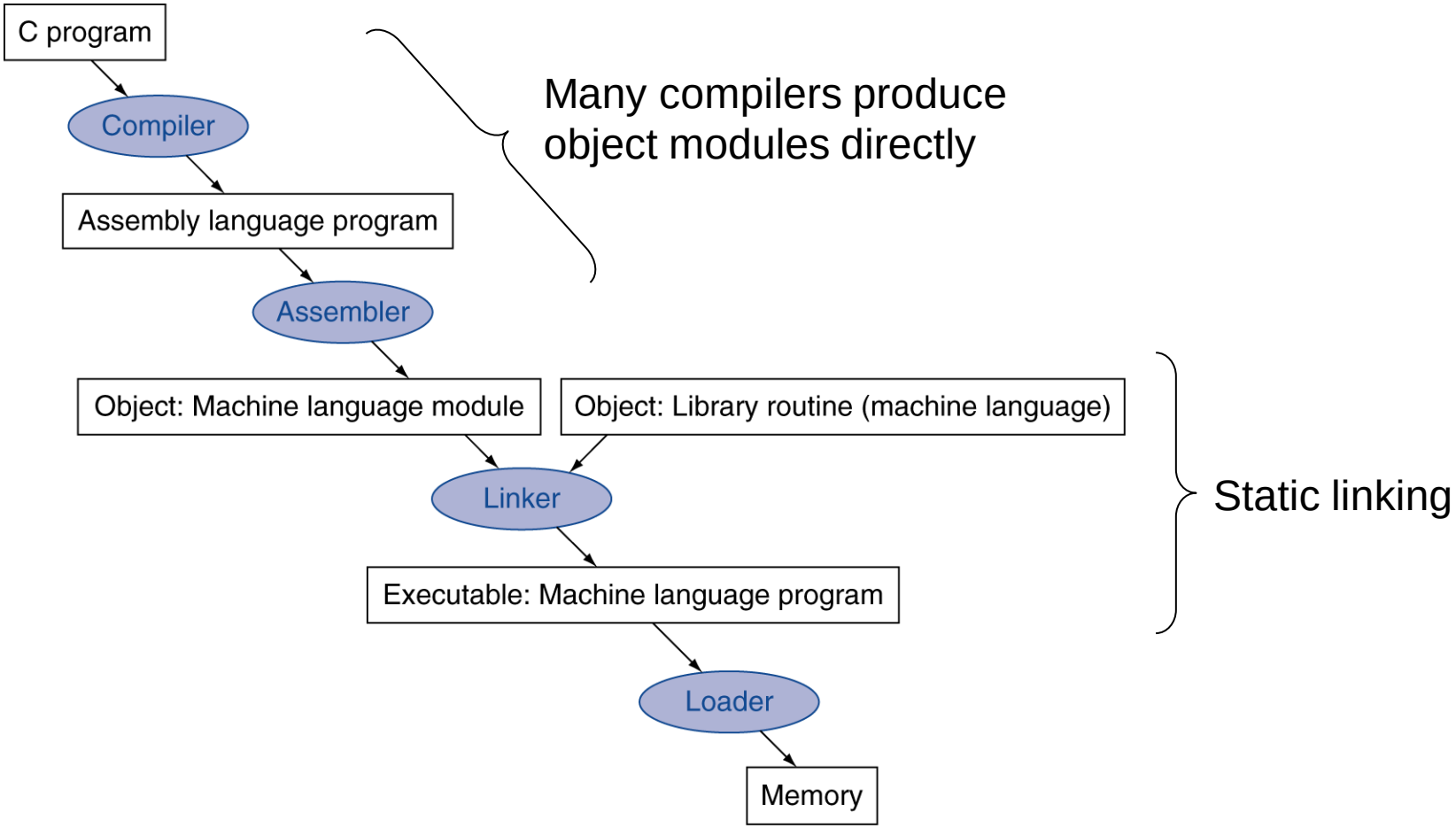


FIGURE B.10.1 MIPS R2000 CPU and FPU.

- Programs development:
  - Writing – coding
  - Loading into computer
  - Execution on computer

- Four steps transforming a C program into a program running on a computer



Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

Example:

- a C-language program

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

FIGURE B.1.5 The routine written in the C programming language.

- Example:
  - An assembly-language program

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra


        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

FIGURE B.1.4 The same routine written in assembly language with labels, but no comments. The commands that start with periods are assembler directives (see pages B-47–49). .text indicates that succeeding lines contain instructions .data indicates that they contain data. .align n indicates that the items on the succeeding lines should be aligned on a *2n* byte boundary. Hence, .align 2 means the next item should be on a word boundary .globl main declares that main is a global symbol that should be visible to code stored in other fi les. Finally, .asciiz stores a null-terminated string in memory.

- Example:
  - An assembly-language program

```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4,  32($29)
sw       $5,  36($29)
sw       $0,  24($29)
sw       $0,  28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8,  $14, 1
slti     $1,  $8, 101
sw       $8,  28($29)
mflo     $15
addu     $25, $24, $15
bne      $1,  $0, -9
sw       $25, 24($29)
lui      $4,  4096
lw       $5,  24($29)
jal      1048812
addiu    $4,  $4, 1072
lw       $31, 20($29)
addiu    $29, $29, 32
jr       $31
move     $2,  $0
```

FIGURE B.1.3 The same routine written in assembly language. However, the code for the routine does not label registers or memory locations nor include comments.

- Example:
  - A machine-language code

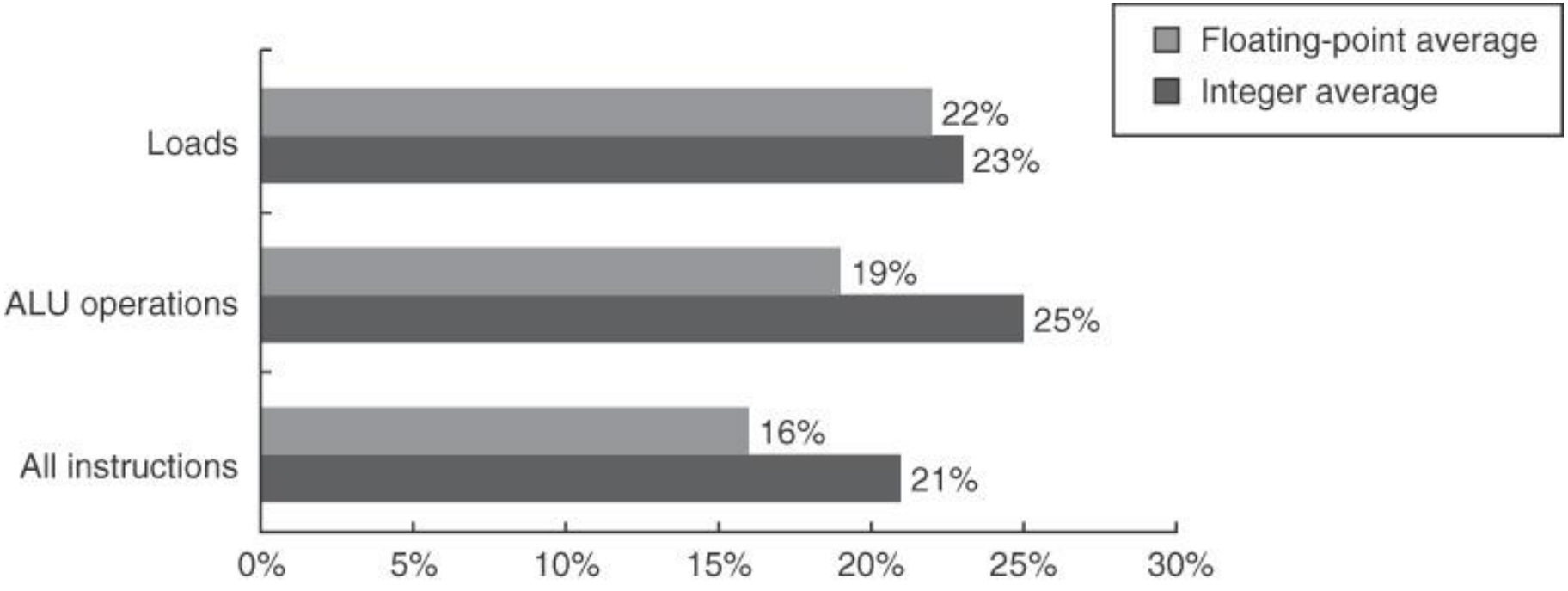
```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

FIGURE B.1.2 MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.

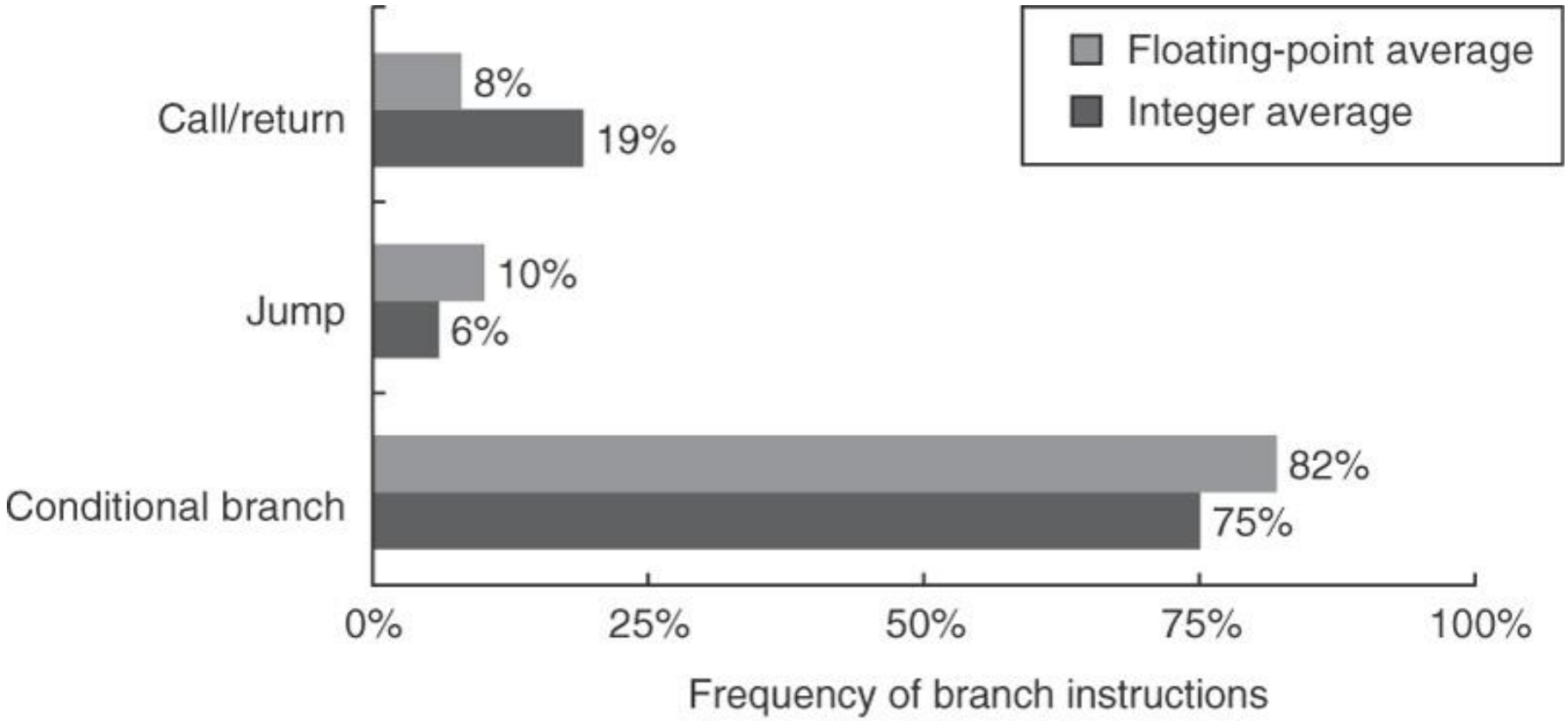
- How to decide on Instructions:
  - Frequency of Usage

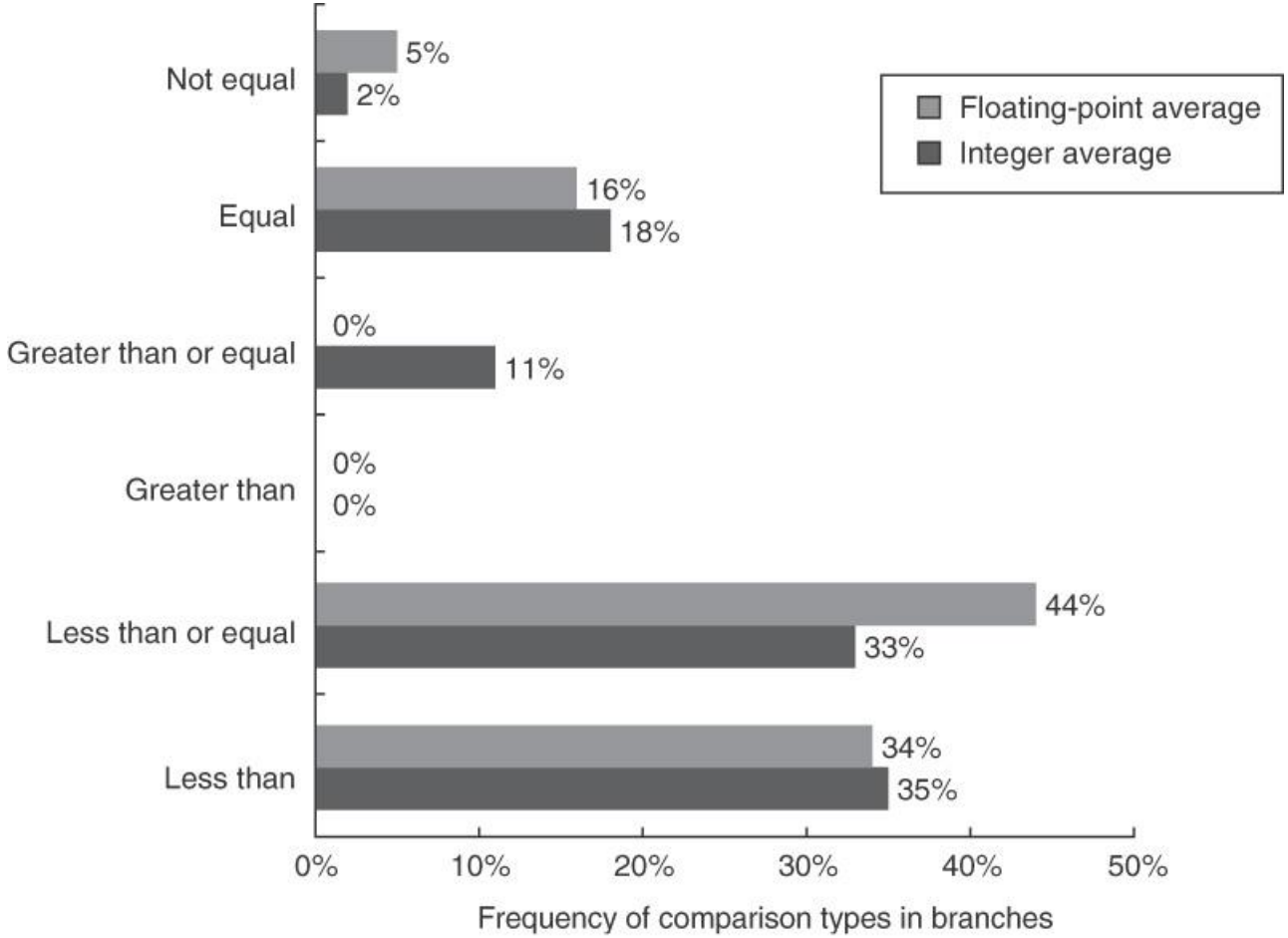
Freq of Immediates

Figure B.9, Page B-12



Lecture 2: Instructions: Language of the Computer



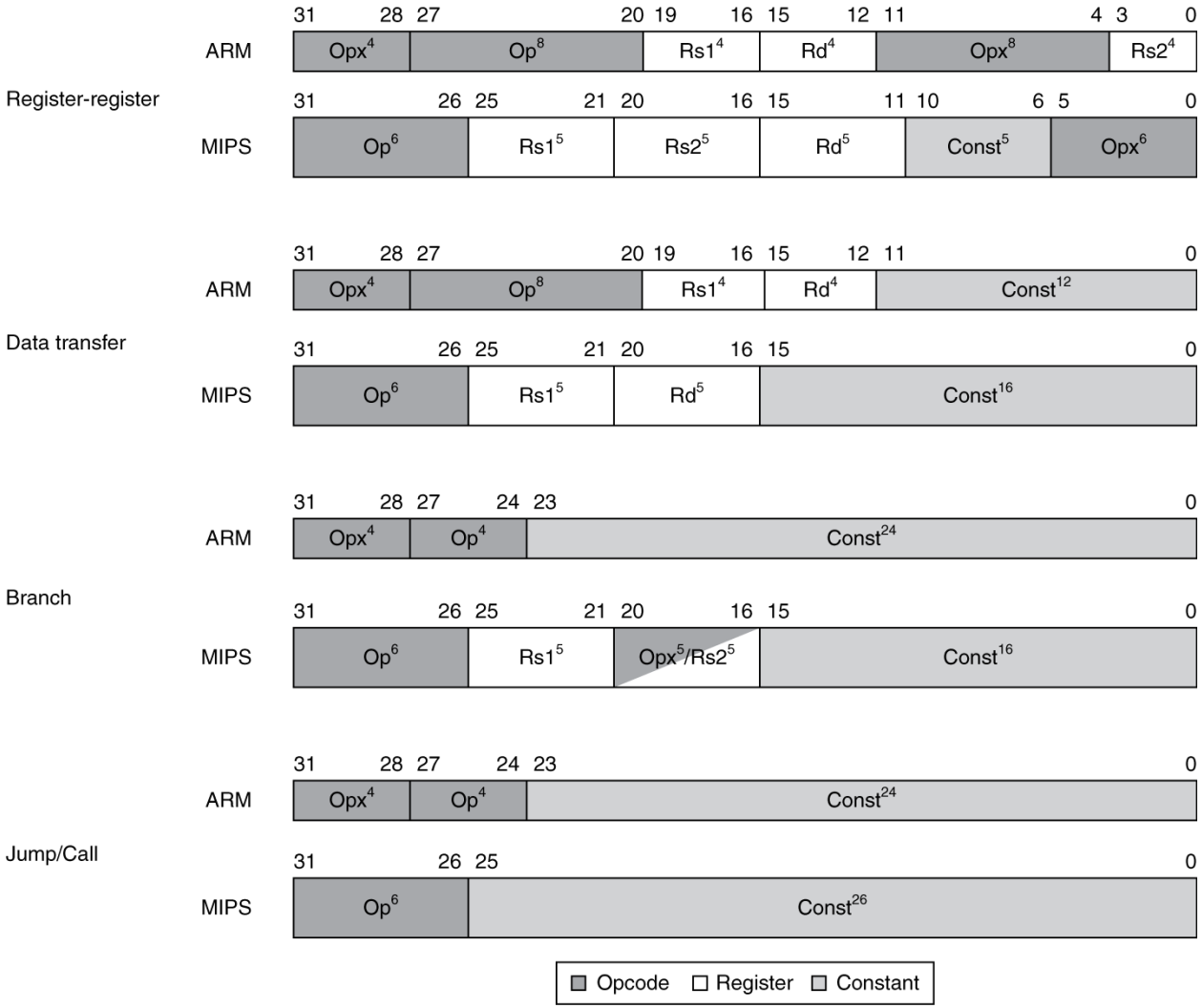


ARMv7 & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

ARMv7 & MIPS Instruction Formats



The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Lecture 2: Instructions: Language of the Computer

Instruction Set Architecture Comparison: x86 vs. MIPS

- General-Purpose Register Architecture:
  - o x86: 16 gprs (general-purpose registers) and 16 fprs (floating-point registers) for register - memory instructions
  - o MIPS: 32 gprs and 32 fprs for load-store instructions
- Memory Accesses:
  - o x86: Byte addressing, non-aligned OK
  - o MIPS: Byte addressing, must be aligned
- Addressing:
  - o x86: Register, Immediate, Displacement, no register (absolute), two-register (based indexed with displacement), two register (based with scaled index and displacement)
  - o MIPS: Register, Immediate, Displacement
- Types and sizes of Operands
  - o x86:8-bit ASCII, 16-bit (unicode character or half word), 32-bit (word or integer), 64-bit (double-word or long-integer), 32-bit single precision and 64-bit double-precision and 80-bit extended double precision floating point operands
  - o MIPS: same as above
- Operations
  - o Both: Data transfer, arithmetic, logic, control, and floating point
- Control Flow Instructions
  - o Both: Conditional branch, unconditional jumps, procedure call and return, (PC-relative addressing)
- Encoding of the instruction
  - o x86: Variable length (1-18 bytes)
  - o MIPS: Fixed, all instructions 32-bit long

Adapted from Prof. David Patterson

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

ARM v8 Instructions

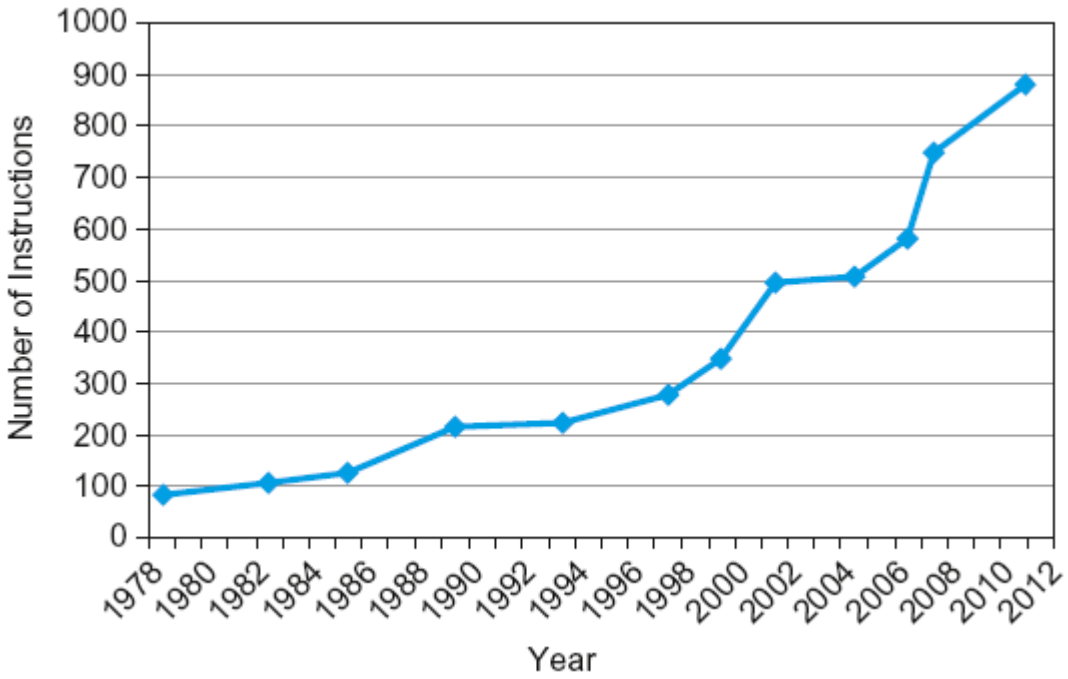
- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks:

- Design principles
  - Simplicity favors regularity
    - √ Fixed size instructions
  - Smaller is faster
    - √ Limited instruction set
  - Good design demands good compromises
    - √ Simple instruction formats
  - Make the common case fast
    - √ Load-store instructions
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - 5 categories of instructions
  - c.f. x86

Concluding Remarks:

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%

- For a programmer
  - A computer is
    - Instruction set
    - Registers available to the programmer
    - Memory Model
    - Data types

- ISA Processor Architecture/Microarchitecture:
  - The instruction set architecture implies programmer-visible instruction set
    - o Boundary between HW and SW
  - The processor micro-architecture is the internal organization of the processor
    - o Processors with different micro-architectures may share the same architecture; i.e., ISA
    - o Five key areas for compatibility:
      - √ Data representation – data formats
      - √ Data storage
      - √ Data access
      - √ Operations on data
      - √ Instruction decoding

- Next lecture
  - Single-Clock Microarchitecture