Chapter 2

Instructions: Language of the Computer

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

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E(on CD)

MIPS Core Instructions

Arithmetic	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
	Move fr. copr. reg.	mfc0 $1,$epc	$1 = $epc	Used to get exception PC
	multiply	mult $2,$3	Hi, Lo = $2 ¥ $3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $2,$3	Hi, Lo = $2 ¥ $3	64-bit unsigned product in Hi, Lo
	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 and remainder
	Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
	Move from Lo	mflo $1	$1 = Lo	Use to get copy of Lo
Logical	and	and $1,$2,$3	$1 = $2 & $3	3 register operands; logical AND
	or	or $1,$2,$3	$1 = $2 I $3	3 register operands; logical OR
	and immediate	and $1,$2,100	$1 = $2 & 100	Logical AND register, constant
	or immediate	or $1,$2,100	$1 = $2 I 100	Logical OR register, constant
	shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
	shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
Data transfer	load word	lw $1,100($2)	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1,100($2)	Memory[$2+100] = $1	Data from register to memory
	load upper imm.	lui $1,100	$1 = 100 x 2^{16}	Loads constant in upper 16 bits

Number Systems

Four Important Number Systems

System	Why?	Remarks
Decimal	Base 10 (10 fingers)	Most used system
Binary	Base 2. On/Off systems	3 times more digits than decimal
Octal	Base 8.Shorthand notation for working with binary	3 times less digits than binary
Hex	Base 16	4 times less digits than binary

Positional Number Systems

- Have a radix *r* (base) associated with them.
- In the decimal system, *r* = 10:
  - Ten symbols: 0, 1, 2, ..., 8, and 9
  - More than 9 move to next position, so each position is power of 10
  - Nothing special about base 10 (used because we have 10 fingers)
- What does 642.391_{10} mean?

$$6 \times 10^{2} + 4 \times 10^{1} + 2 \times 10^{0} \;.\; 3 \times 10^{-1} + 9 \times 10^{-2} + 1 \times 10^{-3}$$

Increasingly +ve powers of radix ← | Radix point ↑ | → Increasingly -ve powers of radix

Positional Number Systems

- What does 642.391_{10} mean?

Radix point

Base 10 (r)	10^2 (100)	10^1 (10)	10^0 (1)	10^{-1} (0.1)	10^{-2} (0.01)	10^{-3} (0.001)
Coefficient (a_j)	6	4	2	3	9	1
Product: $a_j * r^i$	600	40	2	0.3	0.09	0.001
Value	= 600 + 40 + 2 + 0.3 + 0.09 + 0.001 = 642.391					

- Multiply each digit by appropriate power of 10 and add them together
- In general:

$$\sum_{i=-m}^{n} a_j \times r^i$$

Positional Number Systems

Number system	Radix	Symbols
Binary	2	{0,1}
Octal	8	{0,1,2,3,4,5,6,7}
Decimal	10	{0,1,2,3,4,5,6,7,8,9}
Hexadecimal	16	{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}

Binary Number System

Decimal	Binary	Decimal	Binary
0	0000	**8**	1000
1	0001	**9**	1001
2	0010	**10**	1010
3	0011	**11**	1011
4	0100	**12**	1100
5	0101	**13**	1101
6	0110	**14**	1110
7	0111	**15**	1111

Octal Number System

Decimal	Octal	Decimal	Octal
0	0	8	10
1	1	9	11
2	2	10	12
3	3	11	13
4	4	12	14
5	5	13	15
6	6	14	16
7	7	15	17

Hexadecimal Number System

Decimal	Hex	Decimal	Hex
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

Four Number Systems

Decimal	Binary	Octal	Hex	Decimal	Binary	Octal	Hex
0	0000	0	0	8	1000	10	8
1	0001	1	1	9	1001	11	9
2	0010	2	2	10	1010	12	A
3	0011	3	3	11	1011	13	B
4	0100	4	4	12	1100	14	C
5	0101	5	5	13	1101	15	D
6	0110	6	6	14	1110	16	E
7	0111	7	7	15	1111	17	F

Conversion between number systems

Conversion: Binary to Decimal

Binary → Decimal

1101.011_2 → $(??)_{10}$

r	$2^3(8)$	$2^2(4)$	$2^1(2)$	$2^0(1)$	$2^{-1}(0.5)$	$2^{-2}(0.25)$	$2^{-3}(0.125)$
a_j	**1**	**1**	**0**	**1**	**0**	**1**	**1**
a_j*r	**8**	**4**	**0**	**1**	**0**	**0.25**	**0.125**
	$(1101.011)_2 = 8 + 4 + 1 + 0.25 + 0.125 = 13.375$						

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \, . \, 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 13.375_{10}$

↑

Binary point

Conversion: Decimal to Binary

- A decimal number can be converted to binary by repeated division by 2 if it is an integer

number	÷2	Remainder	
155	77	1	Least Significant Bit (LSB)
77	38	1	
38	19	0	
19	9	1	
9	4	1	
4	2	0	
2	1	0	
1	0	1	Most Significant Bit (MSB)

Arrange remainders in reverse order

$155_{10} = 10011011_2$

Conversion: Decimal to Binary

- If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, each part must be converted differently.

Decimal → Binary

$(27.375)_{10}$ → $(??)_2$

number	÷2	Remainder
27	13	1
13	6	1
6	3	0
3	1	1
1	0	1

Arrange remainders in reverse order: 11011

number	X2	Integer
0.375	0.75	0
0.75	1.50	1
0.50	1.0	0

Arrange in order: 011

$\Rightarrow 27.375_{10} = 11011.011_2$

Conversion: Octal to Binary

Octal → Binary

$345.5602_8 \rightarrow (??)_2$

$$\underbrace{3}_{011}\ \underbrace{4}_{100}\ \underbrace{5}_{101}\ .\ \underbrace{5}_{101}\ \underbrace{6}_{110}\ \underbrace{0}_{000}\ \underbrace{2}_{010}$$

$345.5602_8 = 11100101.101110000010_2$

Conversion: Binary to Octal

Binary → Octal

11001110.0101101_2 → $(??)_8$

Note trailing zeros

11 001 110 . 010 110 100

3 1 6 2 6 4

Group by 3's
Add leading zeros if necessary

Group by 3's
Add trailing zeros if necessary

$11001110.0101101_2 = 316.264_8$

Conversion: Binary to Hex

Binary → Hex

$11100101101.1111010111_2 \rightarrow (??)_{16}$

Note trailing zeros

111 0010 1101 . 1111 0101 1100

7 2 D . F 5 C

←	→
Group by 4's Add leading zeros if necessary	Group by 4's Add trailing zeros if necessary

$= 72D.F5C_{16}$

Conversion: Hex to Binary

Hex → Binary

$B9A4.E6C_{16}$ → $(??)_2$

$$\underbrace{1011}_{B}\ \underbrace{1001}_{9}\ \underbrace{1010}_{A}\ \underbrace{0100}_{4}\ .\ \underbrace{1110}_{E}\ \underbrace{0110}_{6}\ \underbrace{1100}_{C}$$

$1011100110100100.111001101100_2$

Conversion: Hex to Decimal

Hex → Decimal

$B63.4C_{16}$ → $(??)_{10}$

16^2	16^1	16^0	16^{-1}	16^{-2}
B (=11)	6	3	4	C (=12)
= 2816 + 96 + 3 + 0.25 + 0.046875 = 2915.296875				

$$11 \times 16^2 + 6 \times 16^1 + 3 \times 16^0 \,.\, 4 \times 16^{-1} + 12 \times 16^{-2} = 2915.296875_{10}$$

Binary Numbers

- How many distinct numbers can be represented by *n* bits?

No. of bits	**Distinct nos.**
1	2 {0,1}
2	4 {00, 01, 10, 11}
3	8 {000, 001, 010, 011, 100, 101, 110, 111}
n	2^n

- Number of permutations double with every extra bit
- 2^n *unique* numbers can be represented by *n* bits

Number System and Computers

- Some tips
  - Binary numbers often grouped in fours for easy reading
  - 1 byte=8-bit, 1 word = 4-byte
  - In computer programs (e.g. Verilog, C) by default decimal is assumed
  - To represent other number bases use

System	Representation	Example for 20
Hexadecimal	0x...	0x14
Binary	0b...	0b10100
Octal	0o… (zero and 'O')	0o24

Number System and Computers

- Addresses often written in Hex
  - Most compact representation
  - Easy to understand given their hardware structure
  - For a range 0x000 – 0xFFF, we can immediately see that 12 bits are needed, 4K locations
  - Tip: 10 bits = 1K

Signed Binary

Negative numbers representation

- Three kinds of representations are common:
  1. Signed Magnitude (SM)
  2. One's Complement
  3. Two's Complement

Signed Magnitude Representation

[0,1] {…………}

Sign bit (left most) — (n -1) magnitude bits

- 0 indicates +ve
- 1 indicates -ve

8 bit representation for +13 is 0 0001101

8 bit representation for -13 is 1 0001101

1's Complement Notation

Let N be an n-bit number and $\tilde{N}(1)$ be the 1's Complement of the number. Then,

$$\tilde{N}(1) = 2^n - 1 - N$$

- The idea is to leave positive numbers as is, but to *represent negative numbers by the 1's Complement of their magnitude*.
- *Example*: Let $n = 4$. What is the 1's Complement representation for +6 and -6?
  - +6 is represented as 0110 (as usual in binary)
  - -6 is represented by 1's complement of its magnitude (6)

1's Complement Notation

- 1's C representation can be computed in 2 ways:
  - *Method 1*: 1's C representation of -6 is:

    $2^4 - 1 - |N| = (16 - 1 - 6)_{10} = (9)_{10} = (1001)_2$

  - *Method 2*: For -6, the magnitude = 6 = $(0110)_2$
    - The 1's C representation is obtained by complementing the bits of the magnitude: $(1001)_2$
    - $2^4 - 1 - |N| = (16)_{10} - 1 - |N| = (15)_{10} - |N|$

      $= (1111)_2 - |N|$

2's Complement Notation

Let N be an n bit number and Ñ(2) be the 2's Complement of the number. Then,

$$\tilde{N}(2) = 2^n - N$$

- Again, the idea is to leave positive numbers as is, but to *represent negative numbers by the 2's C of their magnitude*.
- *Example*: Let $n = 5$. What is the 2's C representation for +11 and -13?
  - +11 is represented as 01011 (as usual in binary)
  - -13 is represented by 2's complement of its magnitude (13)

2's Complement Notation

- 2's C representation can be computed in 2 ways:
  - *Method 1*: 2's C representation of -13 is $2^5 - |N| = (32 - 13)_{10} = (19)_{10} = (10011)_2$
  - *Method 2*: For -13, the magnitude = 13 = $(01101)_2$
    - The 2's C representation is obtained by adding 1 to the 1's C of the magnitude
    - $2^5 - |N| = (2^5 - 1 - |N|) + 1 =$ 1's C + 1

$$01101 \xrightarrow{1's\ C} 10010 \xrightarrow{add\ 1} 10011$$

Comparing all Signed Notations

4-bit No.	SM	1's C	2's C
0000	+0	+0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

- In all 3 representations, a –ve number has a 1 in MSB location
- To handle –ve numbers using *n* bits,
  - ≅ 2^{n-1} symbols can be used for positive numbers
  - ≅ 2^{n-1} symbols can be used for negative umbers
- In 2's C notation, only 1 combination used for 0

Instructions

Arithmetic Operations

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

```
add a, b, c  # a gets b + c
```

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

Register Operands (1)

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file(32-bit data called a “word”), numbered from 0 to 31
  - Use for frequently accessed data

Register Number	Mnemonic Name	Conventional Use		Register Number	Mnemonic Name	Conventional Use
\$0	zero	Permanently 0		\$24, \$25	\$t8, \$t9	Temporary
\$1	\$at	Assembler Temporary (reserved)		\$26, \$27	\$k0, \$k1	Kernel (reserved for OS)
\$2, \$3	\$v0, \$v1	Value returned by a subroutine		\$28	\$gp	Global Pointer
\$4–\$7	\$a0–\$a3	Arguments to a subroutine		\$29	\$sp	Stack Pointer
\$8–\$15	\$t0–\$t7	Temporary (not preserved across a function call)		\$30	\$fp	Frame Pointer
\$16–\$23	\$s0–\$s7	Saved registers (preserved across a function call)		\$31	\$ra	Return Address

Register Operand (2)

- *Design Principle 2:* Smaller is faster
- Example:
  - C code: f = (g + h) - (i + j);
  - MIPS code

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $t2, $t0, $t1
```

	$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
$s0 - $s7		g	h	i	j			

	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7	$t8	$t9
$t0 - $t7	g+h	i+j	final							

Memory Operands (1)

- Main memory used for composite data
  - Arrays, structures, dynamic data
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- Length of an address is 32-bit
  - Min value of address = 0
  - Max value of address = $(2^{32}-1)$
- MIPS is Big Endian
  - Most-significant byte at least address of a word

Address	DATA 32-b
4*N	10101010
…	…
…	…
8	10101010
4	01001110
0	110…0100

Memory Operands (2)

- Data is transferred between memory and register using data transfer instructions: lw and sw

Category	Instruction	Example	Meaning	Comments
Data transfer	load word	`lw $s1,100($s2)`	`$s1 ← memory[$s2+100]`	Memory to Register
	store word	`sw $s1,100($s2)`	`memory[$s2+100]← $s1`	Register to memory

  - $s1 is receiving register
  - $s2 is base address of memory, 100 is called the offset, so ($s2+100) is the address of memory location

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 → 32 ; base register → $s3

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:* Make the common case fast
  - 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
```

Sign 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

Presenting MIPS Instructions in Binary

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

$00000010001100100100000000100000_2 = 02324020_{16}$

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
- Example: Load array A[8] to register $t0, base address of A in $s3

lw $t0, 32($s3)

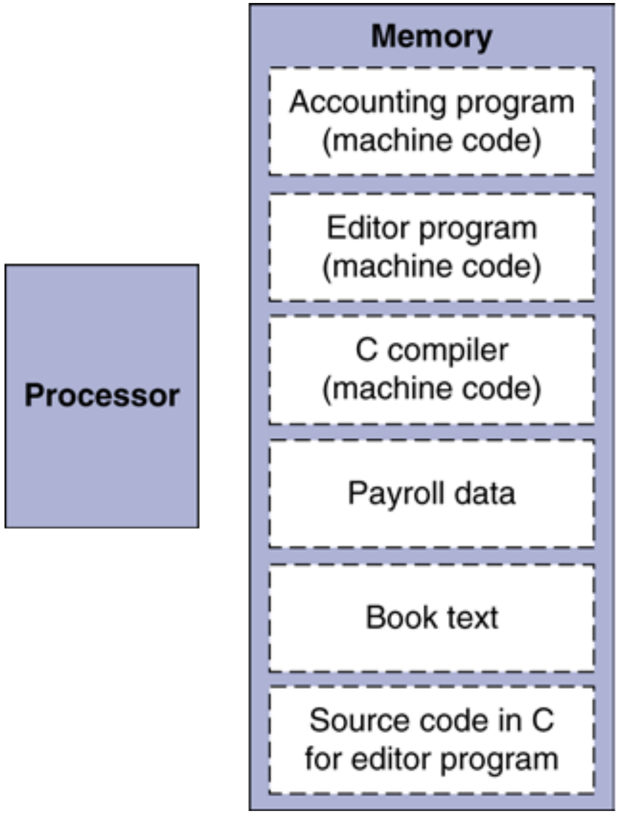
op	**rs**	**rt**	**Constant or address**
35	**9**	**20**	**32**
100011	**01001**	**10100**	**0000,0000,0010,0000**

MIPS I-format Instructions

- *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)

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 0 == NOT (a OR 0) = NOT a
  - Example:

a=0000 0000 0000 0000 0000 0000 1100 1010

```
a is placed in $t1
 nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0000 0000 1100 1010
$t0	1111 1111 1111 1111 1111 1111 0011 0101

Conditional Operations

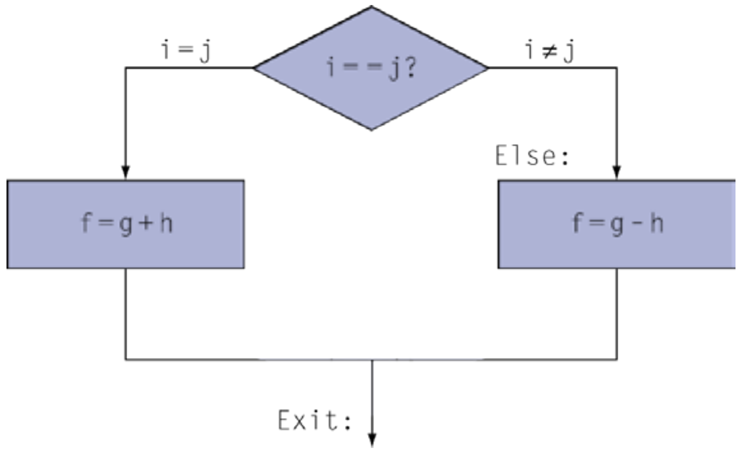
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Example: If Statements

- C code:

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

  - f, g,h,i,j in $s0 ~ $s4



- Compiled MIPS code:

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

Example: 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 # iX4 get offset
      add $t1,$t1,$s6 #get address
      lw   $t0, 0($t1) #$t0=save[i]
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

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

Acknowledgement

- The slides are adapted from Computer Organization and Design, 4th Edition, by David A. Patterson and John L. Hennessy, 2008, published by MK (Elsevier)