CS350/CS402

Chapter 2

Instructions: Language of the Computer

Dr. Yonshik Choi
Illinois Institute of Technology

Revised original document from the publisher.

Languages

- In traveling foreign country, how to buy a product you want?
- Use that country's currency (money).
- To communicate foreigner, use that foreign language is the best way

Instructions

§2.1 Introduction

- To command machine (computer), what to do?
- Use machine language only comes with binary patterns
- Machine language is called *Instructions*
- Instruction set: vocabulary of commands understood by a given architecture

E.g. Assembler: symbolic representation

Instruction Set

§2.1 Introduction

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common

Why?

- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

§2.1 Introduction

Instruction Set

- The repertoire of instructions of a

- *All computers are based on similar principles*
- *Common basic operations all computers provide*
- *Common goal of hardware designer*
  - *Max performance, min cost and power*
- *Simplicity of an equipment*

e

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (now Imagination Technologies)
- 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

MIPS History

- Founded by Stanford Univ. researchers in 1984
- SGI used MIPS design computer in 1988
- DEC built a UNIX workstation with MIPS in 1989
- SGI acquired MIPS to secure its supply and named MIPS Technologies in 1992
- Sony Playstation 2 with MIPS R5900 cpu in 1999
- 2013, MIPS Technologies was sold to Imagination Technologies

Arithmetic Operations

- Every computer must be able to perform arithmetic
- Arithmetic operations are basic function of any computer
- What’s difference between human’s arithmetic operation method and machine?
- High level languages follows human way

  a + b = c   (people use)

  C = A + B (HL)

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 favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Operations

§2.2 Operations of the Computer Hardware

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

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

- All arithmetic operations have this form
- In MIPS arithmetic instruction, only one operation with three variables
- No more than, no less than three variables allowed. Why?

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 favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

§2.2 Operations of the Computer Hardware

MIPS Arithmetic

§2.2 Operations of the Computer Hardware

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

```
C code:     a = b + c + d;

MIPS code:        add a, b, c
                  add a, a, d
```

Arithmetic Example

- C code:

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

- Compiled MIPS code (conceptual example):

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

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
  - c.f. main memory: millions of locations

MIPS Registers

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

A very large number of registers increase the clock cycle time taking electronic signals when they must travel farther.

Previous Example Correction

- C code:

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

- Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

- ***MIPS must use registers instead of variables***
- ***Why? Variables locate in main memory***

Register Operand Example

- C code:

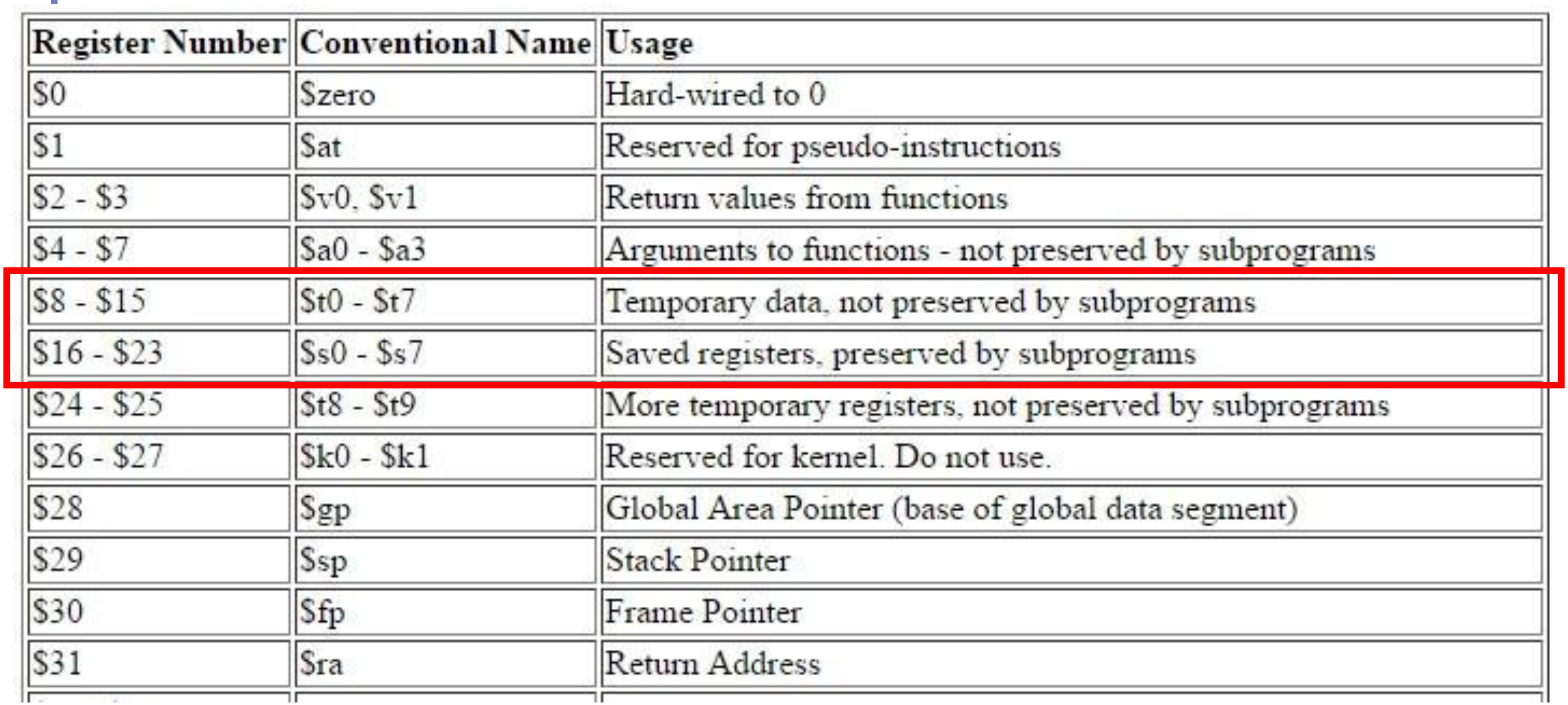
```
f = (g + h) - (i + j);
```

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

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

MIPS registers



Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$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 registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for 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

MIPS example

- C code:

```
sum = a + b;
```

- MIPS to above C code (still conceptual):

```
lw $t0, a            # Load a from memory into a CPU register
lw $t1, b            # Load b from memory into a CPU register
add $t0, $t0, $t1    # Add a and b
sw $t0, sum          # Store the result from the CPU register to memory
```

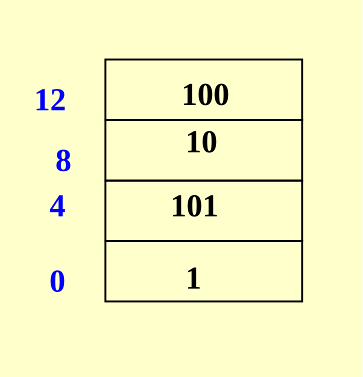
Main Memory

Memory Operands

- Programming languages use simple variables that contain single data elements as well as more complex data structures such as arrays and structures
- Composite data
  - Arrays, structures, dynamic data
- Number of registers in a processor is short
- Composite data can contain more data elements than registers in a processor
  - How can a computer represent and access those large structures?
  - Data structures are kept in memory

Memory Operands

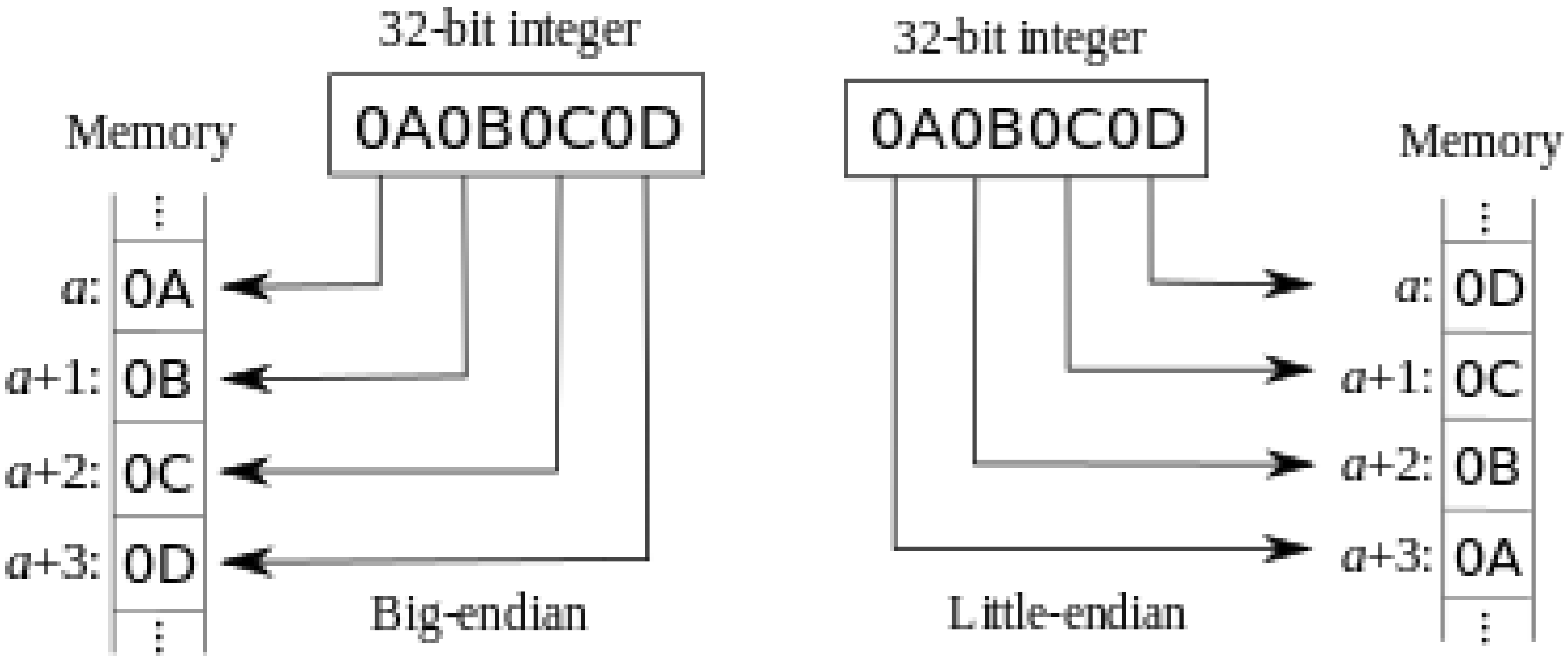
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
  - Arithmetic operations occur only on registers in MIPS
  - So MIPS has instructions data transfer between memory and registers -> **`data transfer instructions`**
- Memory is byte addressed
  - 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



Big Endian vs. Little Endian

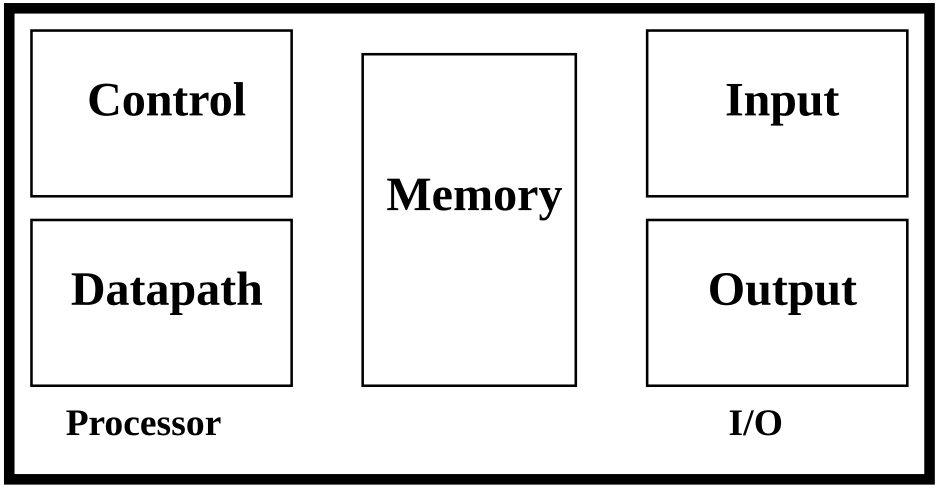
- A matter of how to store/access bits in a memory
- Writes a decimal number 123 on a paper and read it from left to right
- Or writes 321 and read it from right to left
- Why?
- Right positioned bit is the least significant bit (LSB) and left-most bit is the most significant bit (MSB) in binary representation

Big Endian vs. Little Endian



Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided



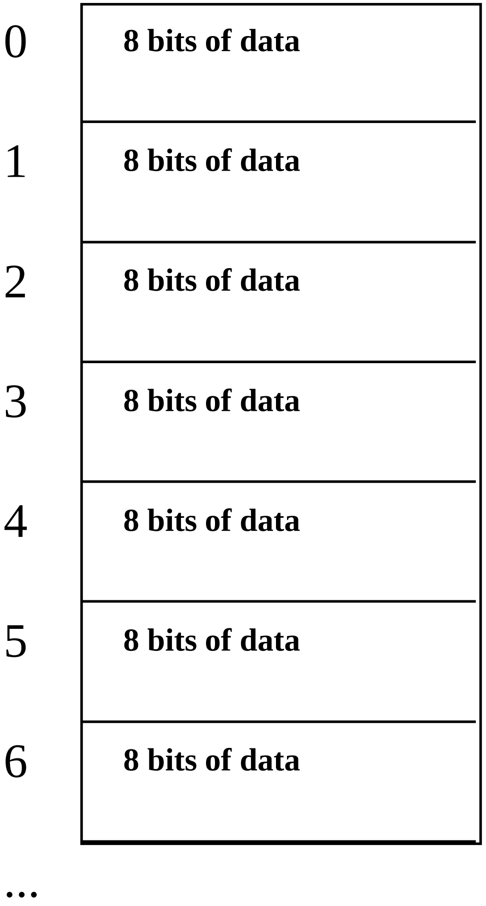
Registers vs. Memory

- Compiler associates variables with registers
- What about programs with lots of variables or complex data structures like arrays and structures?
- HL program defines a lot variables to use
- CPU only has only limited storages

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

Memory Organization



0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

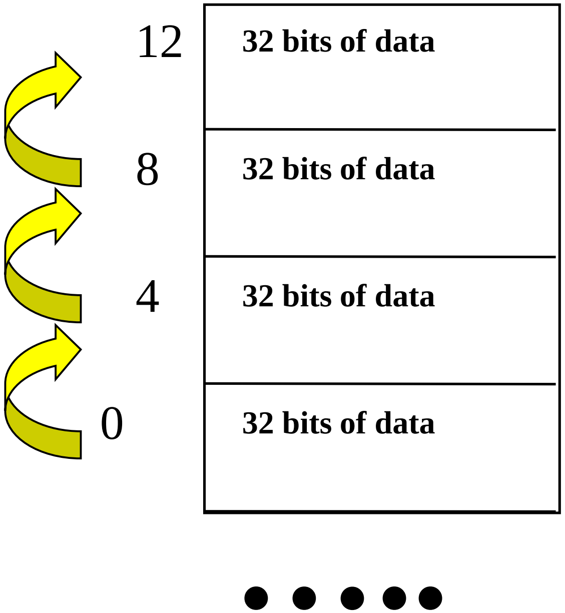
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.
- 2^{32} bytes with byte addresses

  from 0 to 2^{32}-1
- 2^{30} words with byte addresses

  0, 4, 8, ... 2^{32}-4

Memory Organization



12	32 bits of data
8	32 bits of data
4	32 bits of data
0	32 bits of data

• • • • •

Registers hold 32 bits of data

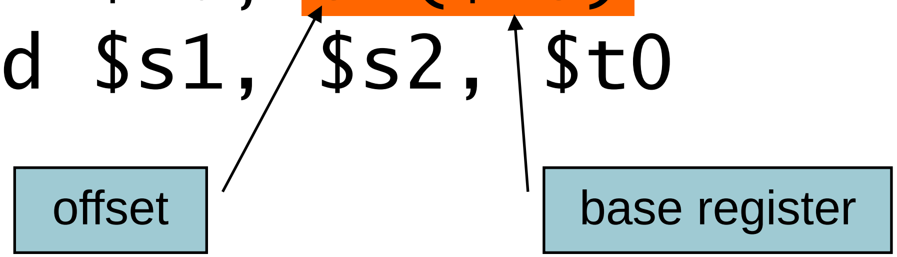
Memory Operand Example 1

- 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
```



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
```

Memory Operand Example 2

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

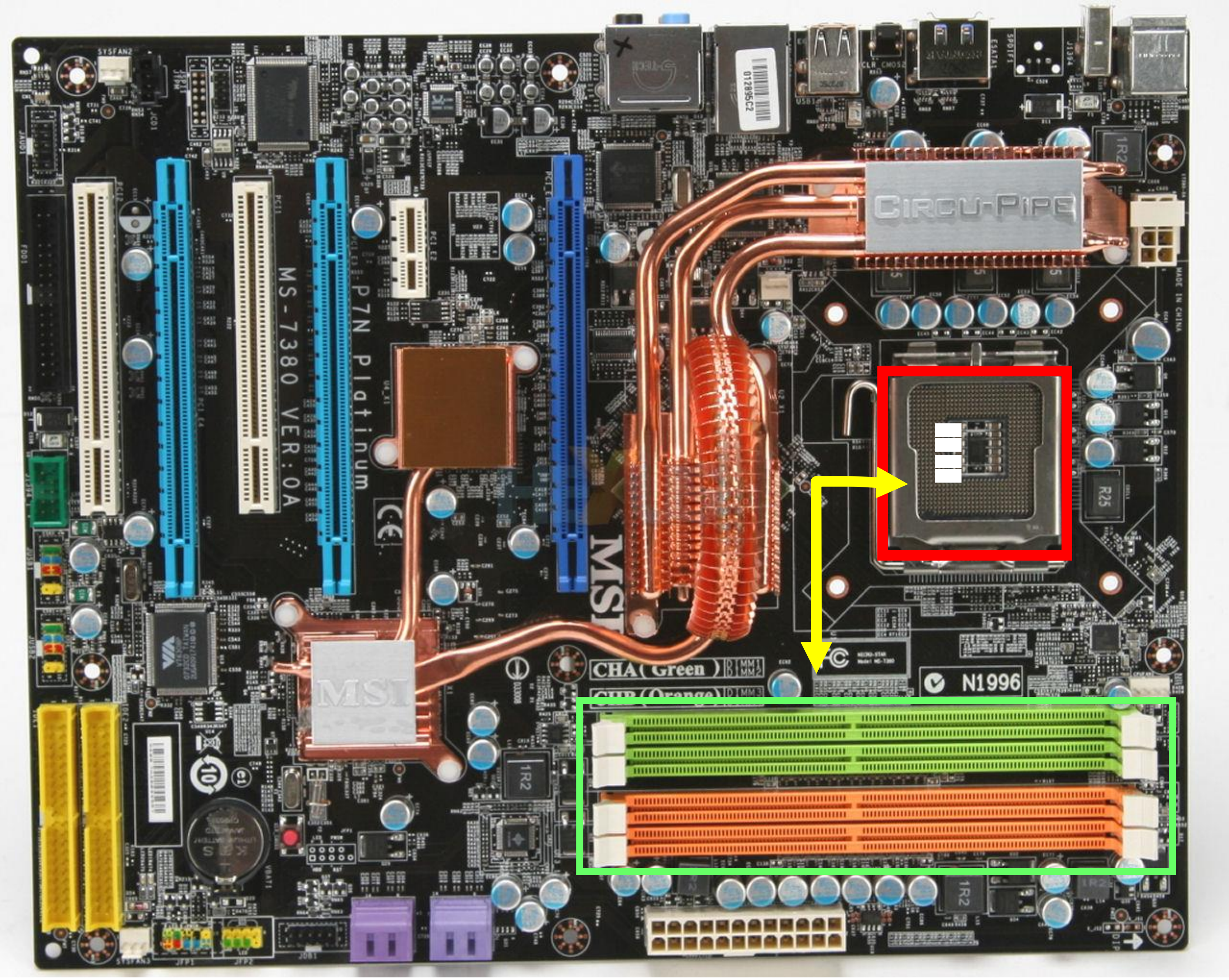
- Can refer to registers by name (e.g., $s2, $t2) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can’t write: `add 48($s3), $s2, 32($s3)`

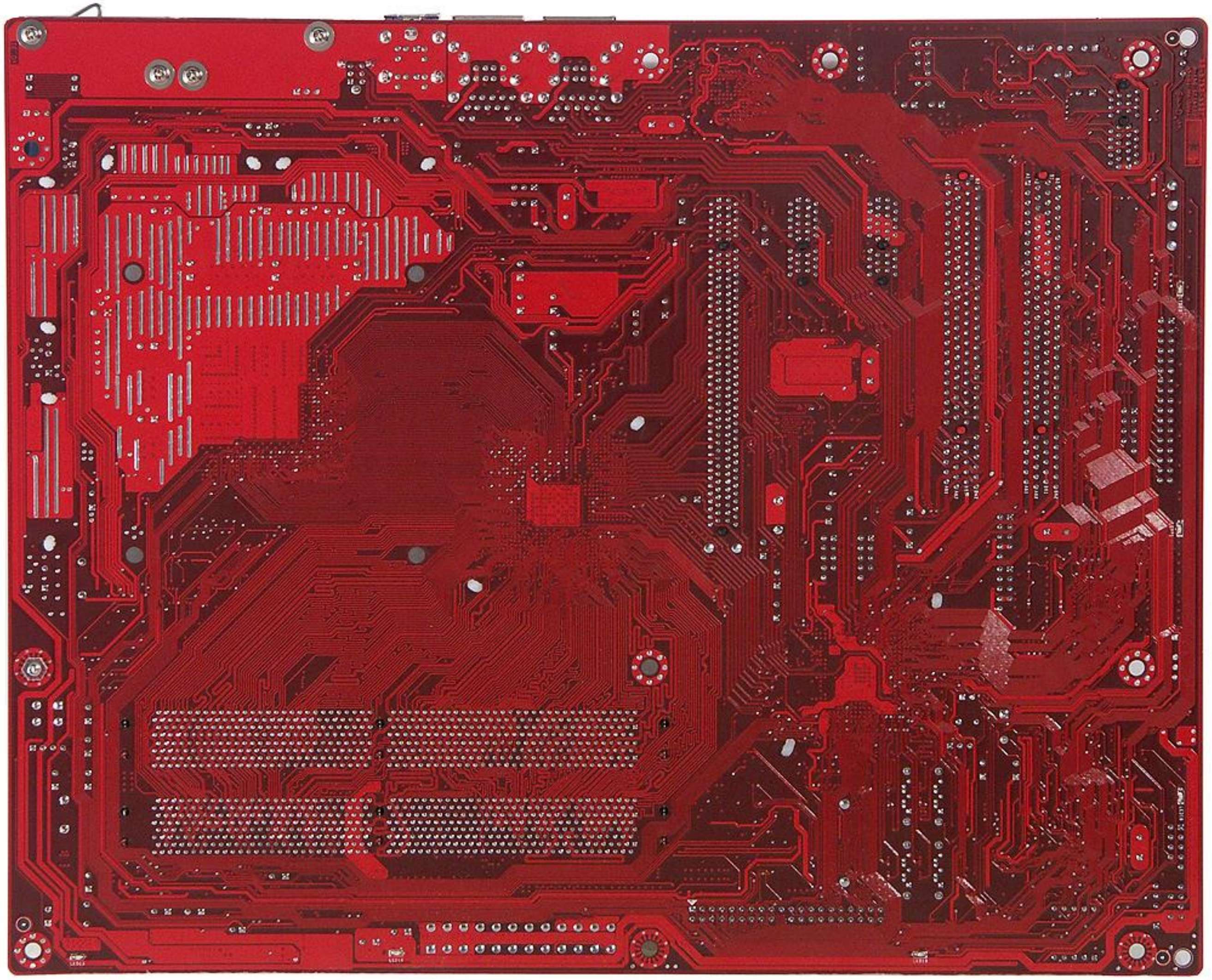
Registers vs. Memory

- 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!

Motherboard



Motherboard



Review

- MIPS
  — loading words but addressing bytes
  — arithmetic on registers only

-

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Review

- HL programs have more variables than computers have registers.
- Compiler keeps the most frequently used variables in registers and remains in memory.

Review

- *Spilling registers*: process of putting less commonly used variables (or needed later) into memory
- Data is more useful in register:
  - arithmetic instructions read two registers, operate, write in a register
  - Data transfer instructions: read or write one operand w/o operating

Immediate Operands

we need to load value of constant from memory to register

```
 lw $t0, AddrConstant4($s1) # $t0 = constant 4
add $s3, $s3, $t0           # $s3 = $s3 + $t0
```

- **Assume AddrConstant4 is the memory address of the constant 4**

or alternatively,

- Constant data specified in an instruction

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

Immediate Operands

- No subtract immediate instruction
  - Just use a negative constant

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

 `Use` *`add immediate`*

- *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
```

Decimal to Binary

- Integer number 123 means 123_{10}
- Convert decimal value to binary value means 123_{10} to 1111011_2
- $123 = 64 + 32 + 16 + 8 + 2 + 1$

$$= 1x2^6 + 1x2^5 + 1x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$$

$$= 2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 1$$

$$= 1111011_2$$

Decimal to Binary

- ***What's the binary value of integer 200?***

Practice manual manners..

11001000_2

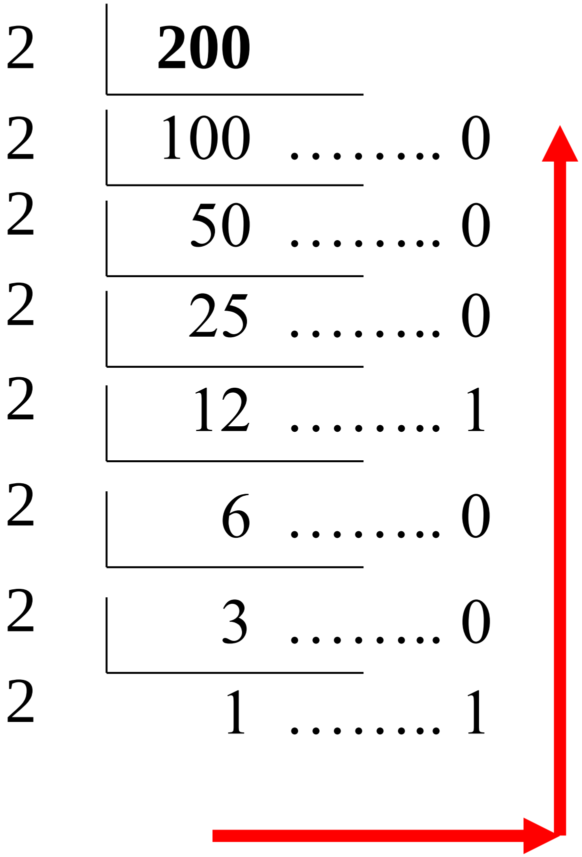
Binary value: m/c understands

Hexadecimal: format between binary representation and human way

Decimal to Binary

- *200* =

Quick manual way



```
2 | 200
2 | 100 ........ 0
2 |  50 ........ 0
2 |  25 ........ 0
2 |  12 ........ 1
2 |   6 ........ 0
2 |   3 ........ 0
2     1 ........ 1
```

11001000_2

Numbers

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: 0...2^n-1

Numbers

- Of course it gets more complicated:

  numbers are finite (overflow)

  fractions and real numbers

  negative numbers

  e.g., no MIPS subi instruction;

  addi can add a negative number

Numbers

How do we represent negative numbers?

i.e., which bit patterns will represent which numbers?

§2.4 Signed and Unsigned Numbers

Numbers

§2.4 Signed and Unsigned Numbers

- Decimal numbers (base 10) vs. Binary numbers (base 2)
- Decimal integer 13 can be represented with binary number like:
  - 13 = 1101
  - MIPS uses 32 bits binary number so,
  - 0000 0000 0000 0000 0000 0000 0000 1101

MSB (Most Significant Bit)
Bit position 31

LSB (Least Significant Bit)
Bit position 0

Different Representation

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

Issues: balance, number of zeros, ease of operations

Which one is best? Why?

Unsigned Binary Integers

- Given an n-bit number

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

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

§2.4 Signed and Unsigned Numbers

Limited values

- Computer cannot recognize infinite values
- Overflow is expected: why?
- Example of using 4 bits system
  - Increase number by 1

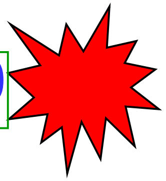
16 numbers
= 2^4

0000
0001
0010
0011
0100
0101
0110
0111

1000
1001
1010
1011
1100
1101
1110
1111

Then what is next?

10000



Overflow from this 4 bit system

2s-Complement Signed Integers

- Given an n-bit number

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

- 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 Integers

- Bit 31 is a sign bit ← MSB
  - 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

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

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

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $–2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Signed Negation

- Again, two steps are required to negate
- Step 1: invert binary digits
- Step 2: add 1

- Example: negate +5 in four digit sys

  $+5 = 0101_2$

  step 1: 1010_2

  step 2: $1010_2 + 1_2 = 1011_2$

  $-5 = 1011_2$

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

MIPS

- 32 bit signed numbers:

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

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

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

...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ ***maxint***

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$ ***minint***

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$

...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$

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

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

- Why max positive integer number is not the same as max negative?

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

Register Usage

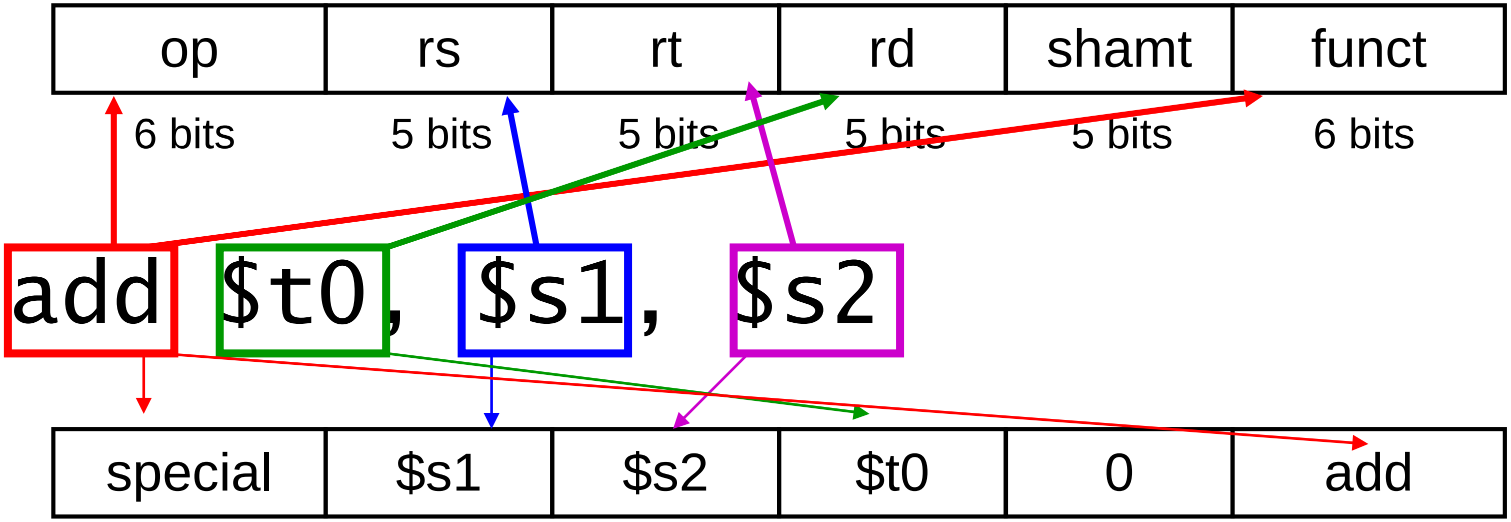
- $v0, $v1: result (return) values (reg’s 2 and 3)
- $a0 – $a3: arguments (reg’s 4 – 7)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg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

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



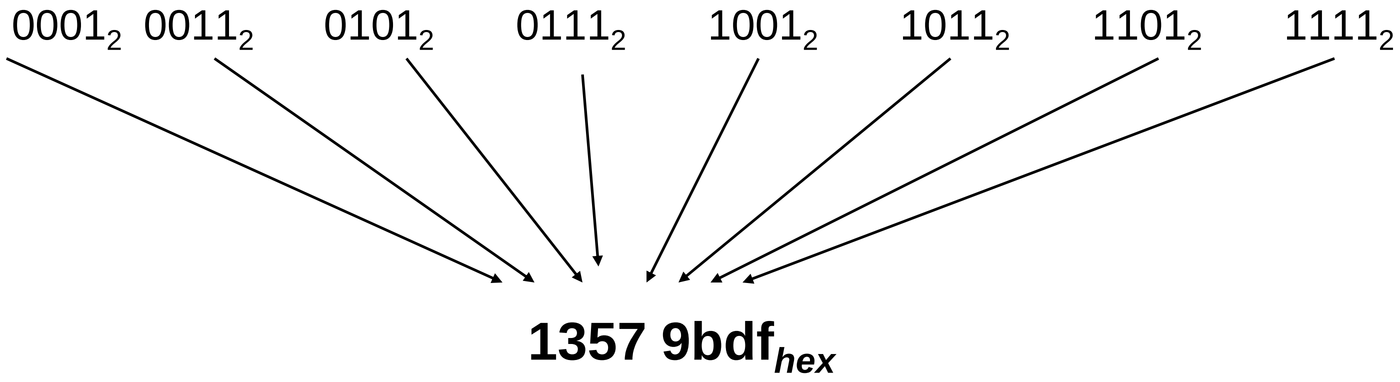
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
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



0001_2 0011_2 0101_2 0111_2 1001_2 1011_2 1101_2 1111_2

1357 9bdf*hex*

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

MIPS R-format Issue?

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

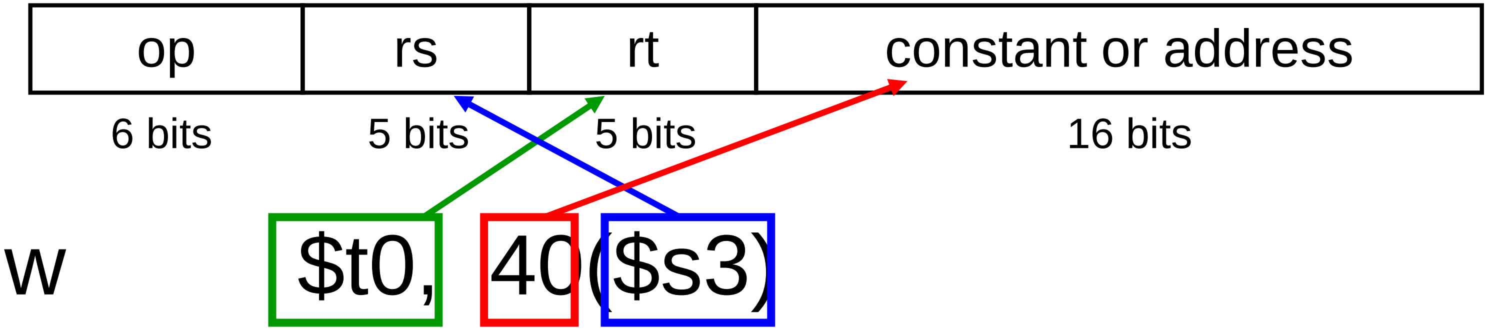
- What if an instruction needs longer fields than 5 bits?
- Instruction “load word” requires two registers.
- 5 bit represents range of 32 ⇒ 32 bytes?
- Instruction addi only use value of 32

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

MIPS I-format Instructions



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

- lw $t0, 40($s3)
  - $t0 or 8 is placed in the rt field
  - $s3 or 19 is placed in the rs field
  - 40 is placed in the address field

  - So 40($s3) is A[10] (if the array name is A)

MIPS I-format Instructions

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

In HL: … + A[10] ⇒ lw $t0, 40($S3)

35	19	8	40
6 bits	5 bits	5 bits	16 bits

100011	10011	01000	0000000000101000
6 bits	5 bits	5 bits	16 bits

10001110011010000000000000101000

Translation

- Example: A[300] = h + A[300];

  if $t1 has the base of the array A
  $s2 corresponds to h

```
lw   $t0, 1200($t1)          # temporary reg $t0 gets A[300]
add  $t0, $s2, $t0           # temporary reg $t0 gets h + A[300]
sw   $t0, 1200($t1)          # stores h + A[300] back into A[300]
```

MIPS m/c lang,

op	rs	rt	rd	add/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

Translation

MIPS m/c lang (decimal value),

op	rs	rt	rd	add/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

Real format (with field tab)

100011	01001	01000		0000 0100 1011 0000	
000000	10010	01000	01000	00000	100000
101011	01001	01000		0000 0100 1011 0000	

10001101001010000000010010110000

00000010010010000100000000100000

10101101001010000000010010110000

Translation

MIPS m/c lang (decimal value),

op	rs	rt	rd	add/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

Real format (with field tab)

100011		01001	01000		0000 0100 1011 0000	
000000		10010	01000	01000	00000	100000
101011		01001	01000		0000 0100 1011 0000	

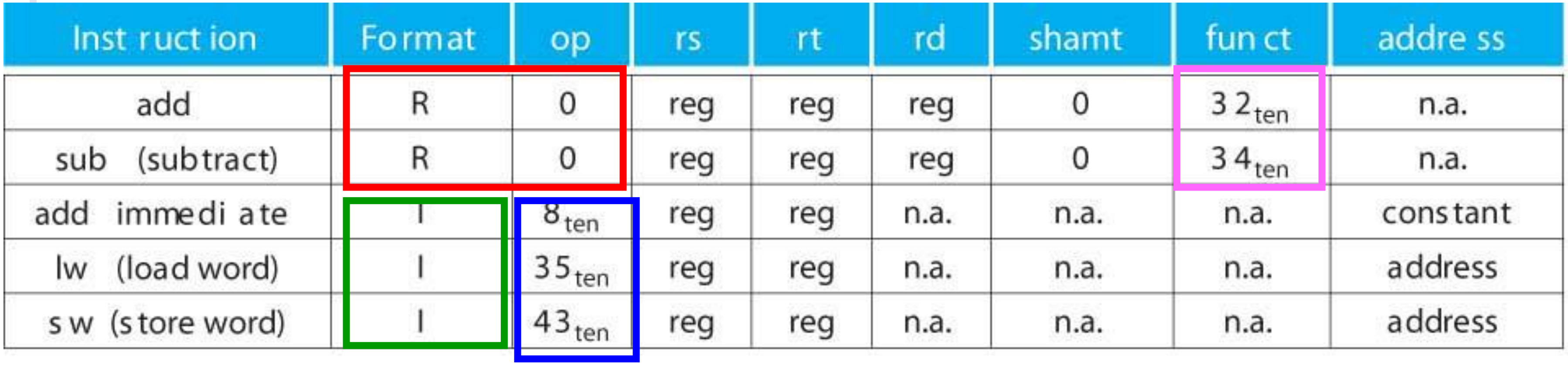
10001101001010000000010010110000

00000010010010000100000000100000

10101101001010000000010010110000

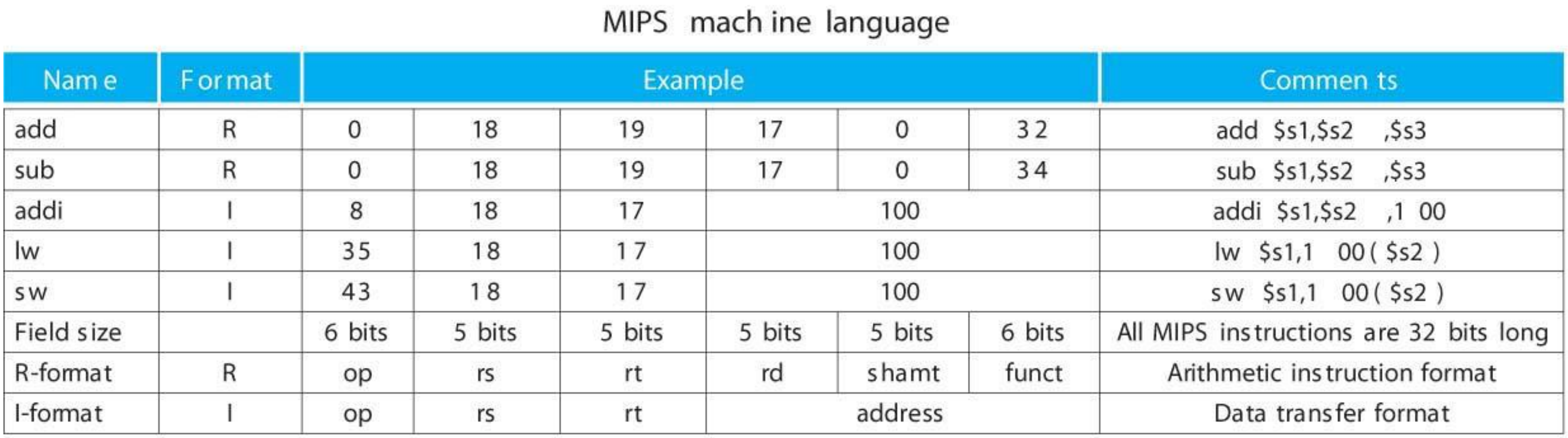
How to know which type of instruction?

- The first field, opcode, tells the differences.



Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

MIPS Machine Language



MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Example

0	8	9	10	0	34

- What is MIPS code of above?

Answer

0	8	9	10	0	34

- What is MIPS code of above?

Answer

0	8	9	10	0	34

- What is MIPS code of above?
  - sub $t2, $t0, $t1
- What about above are hex codes?
  - The answer is NO.
  - Why? Check value carefully.

Stored Program Computers

- Today’s computers are built on two key concepts:

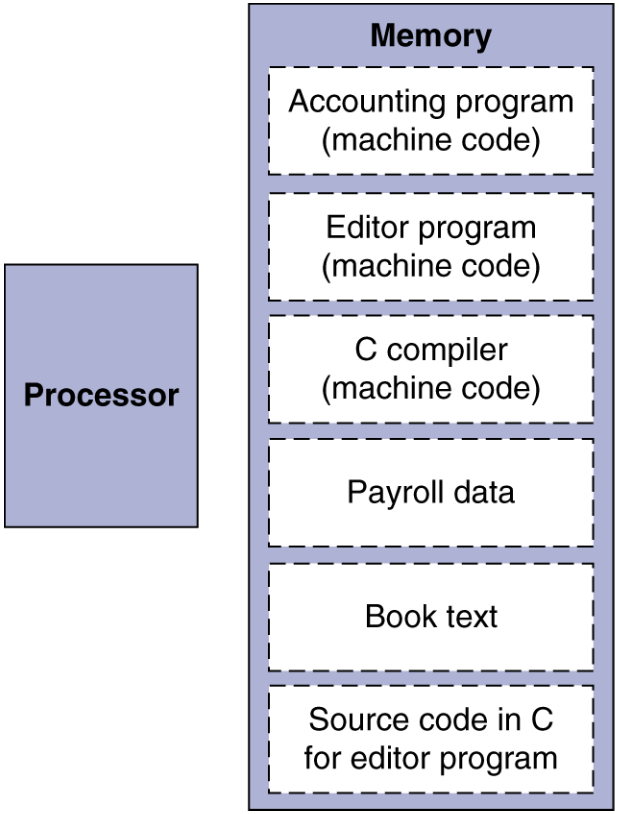
  1. Instructions are represented as numbers

  2. Programs are stored in memory to be read or written, just like numbers

- Binary Compatibility is very important to run a software on a different machine

Stored Program Computers

The BIG Picture


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- 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

What does computer do?

A computer needs to do:

- interpret instructions in order to decide how to
- manipulate the data
- manipulate data in accordance with instructions
- decide what instruction to execute next
- store data (and instruction) words in memory
- retrieve data (and instruction) words from memory
- move words from place to place

Logical Operations

§2.6 Logical Operations

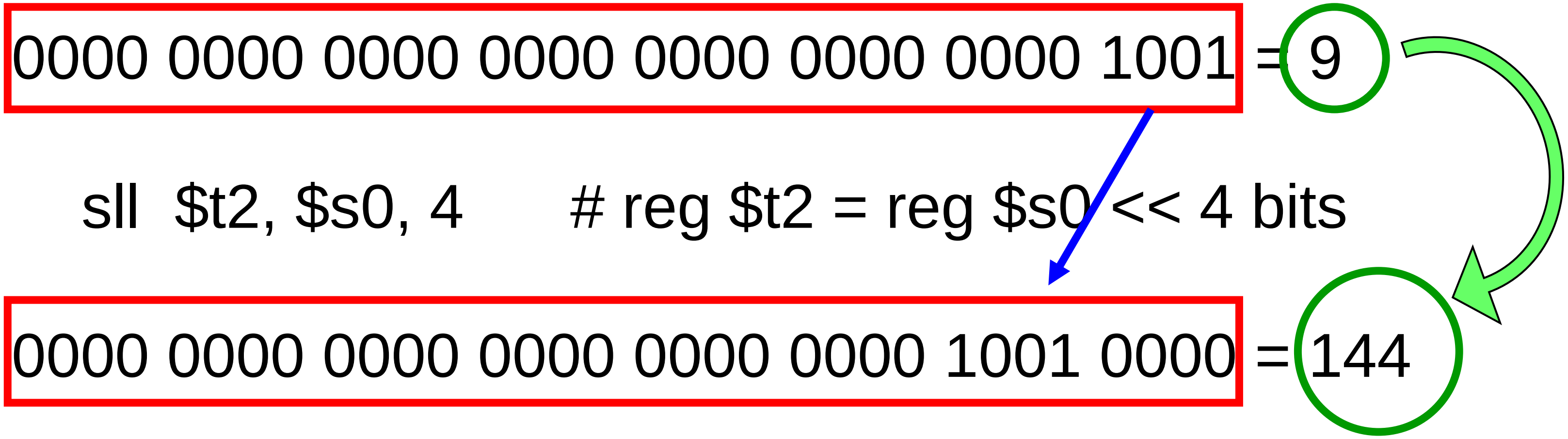
- Instructions for bitwise manipulation

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

sll

- sll (shift left logical)



0000 0000 0000 0000 0000 0000 0000 1001 = 9

sll $t2, $s0, 4 # reg $t2 = reg $s0 << 4 bits

0000 0000 0000 0000 0000 0000 1001 0000 = 144

result: multiply by 2^i

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)

Fill in example

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- sll $t2, $s0, 4

Fill in example (decimal answer)

0	0	16	10	4	0
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- sll $t2, $s0, 4
- opcode: 0
- register rd: 10 ($t2)
- register rt: 16 ($s0)
- register rs: not used
- shift amount: 4 bits
- funct: 0

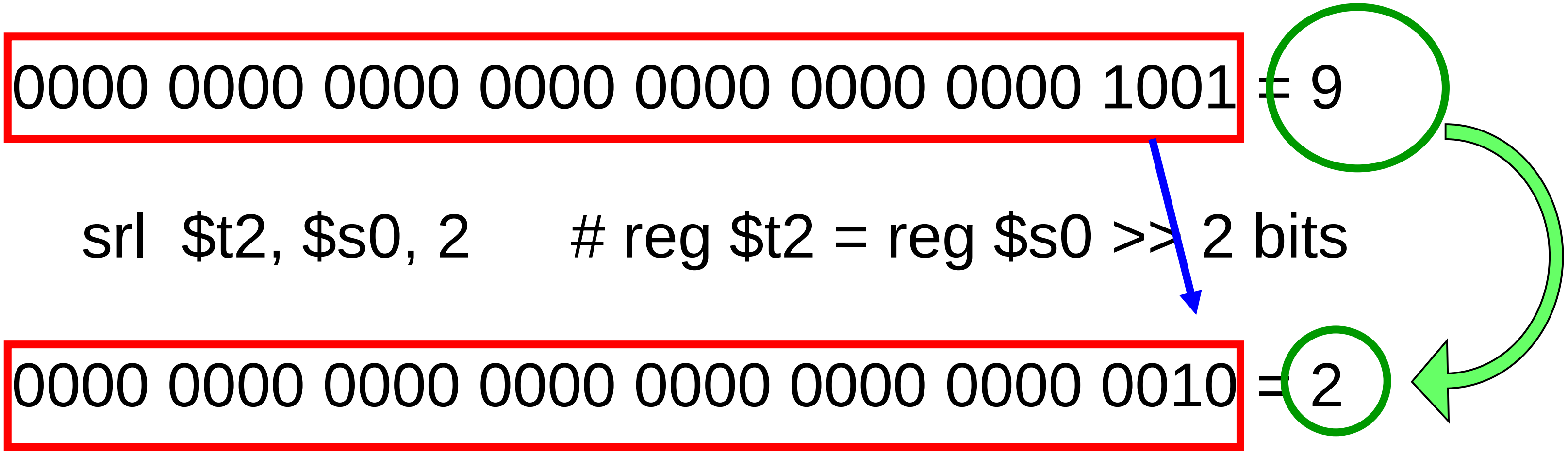
Fill in example (binary answer)

000000	00000	10000	01010	00100	000000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- sll $t2, $s0, 4
- opcode: 0
- register rd: 10 ($t2)
- register rt: 16 ($s0)
- register rs: not used
- shift amount: 4 bits

srl

- srl (shift right logical)



0000 0000 0000 0000 0000 0000 0000 1001 = 9

srl $t2, $s0, 2 # reg $t2 = reg $s0 >> 2 bits

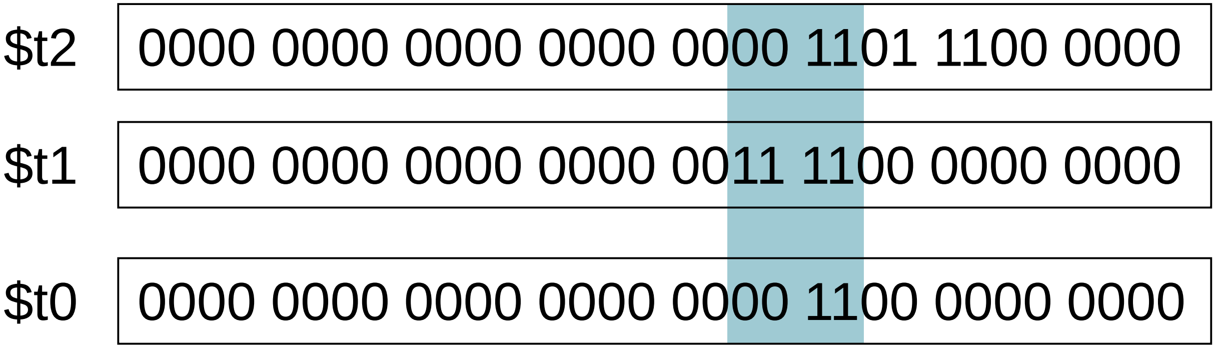
0000 0000 0000 0000 0000 0000 0000 0010 = 2

result: divide by 2^i (integer value)

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

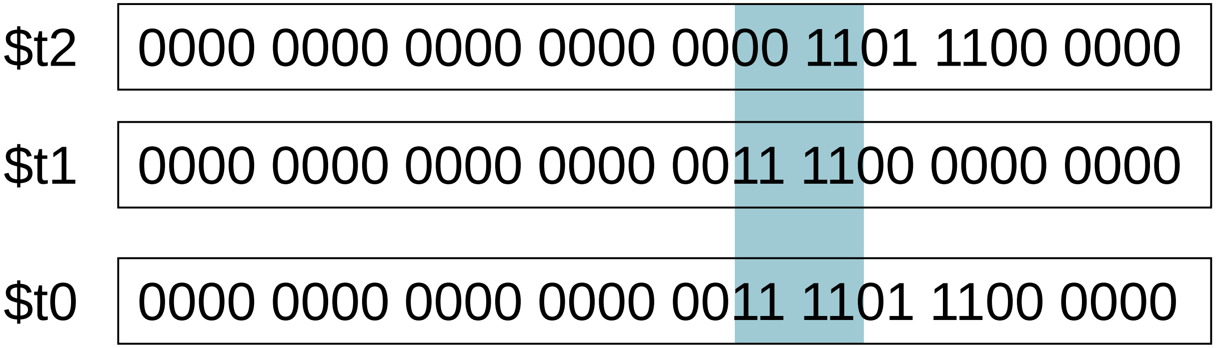
$t1 0000 0000 0000 0000 0011 1100 0000 0000

$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-operands instruction)
  - a NOR b == NOT (a OR b)

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

Register 0: always read as zero

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

Conditional Operations

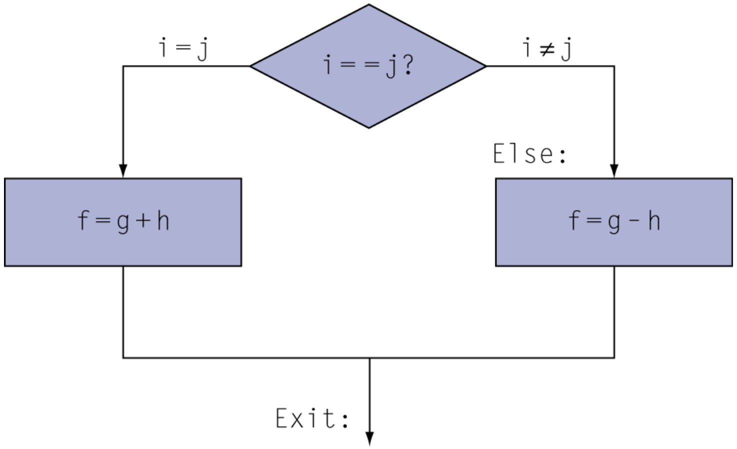
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- High level code:

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

  - f, g, … in $s0, $s1, …

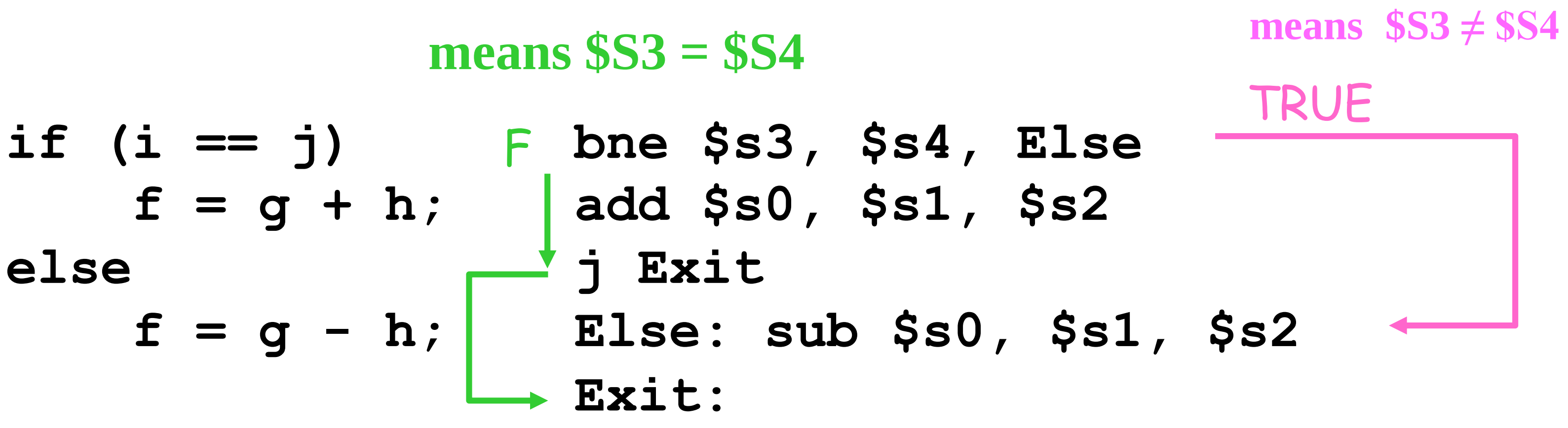


- Compiled MIPS code:

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

Assembler calculates addresses

bne

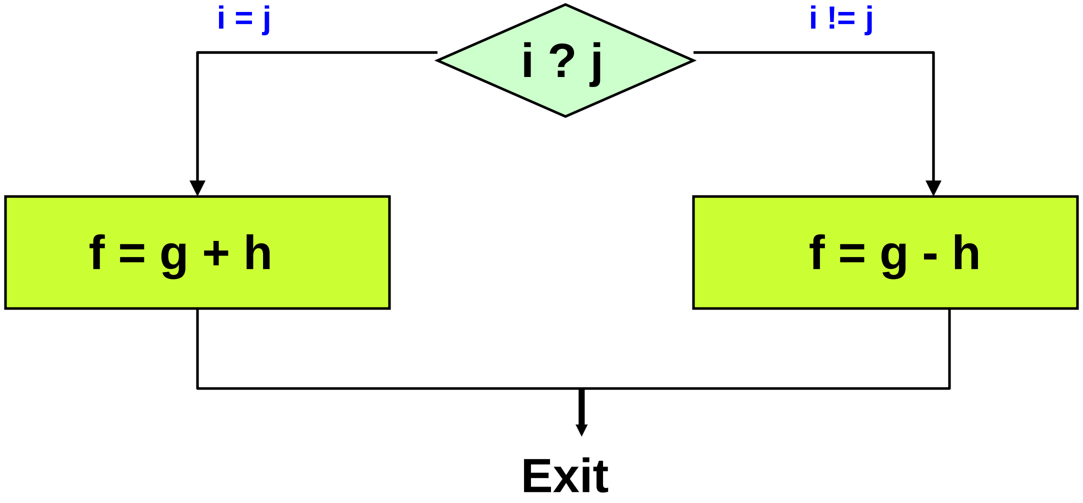


means $S3 = $S4

means $S3 ≠ $S4

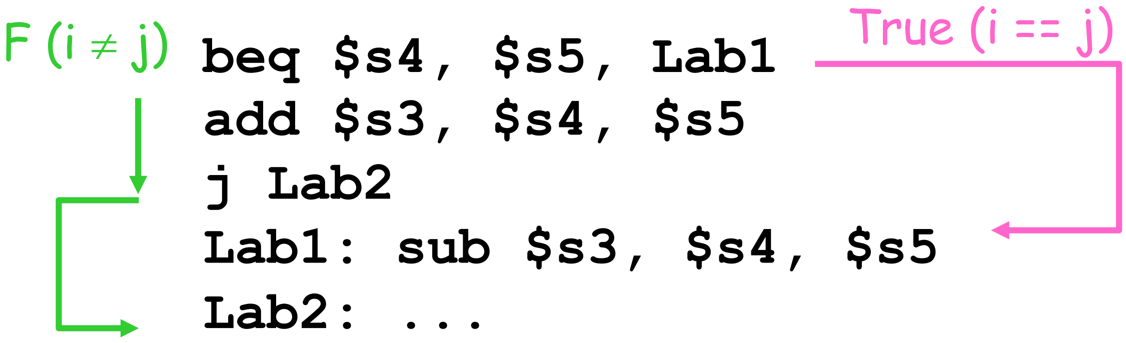
```
if (i == j)
      f = g + h;
else
      f = g - h;
```

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



beq

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



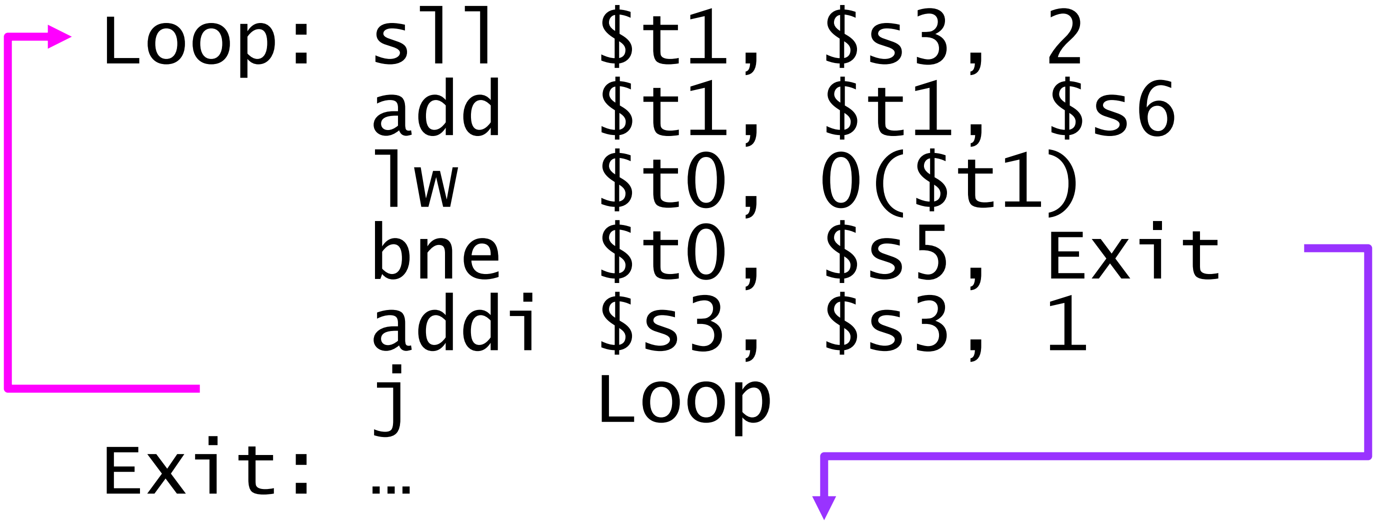
```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

Compiling Loop Statements

- C code:

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

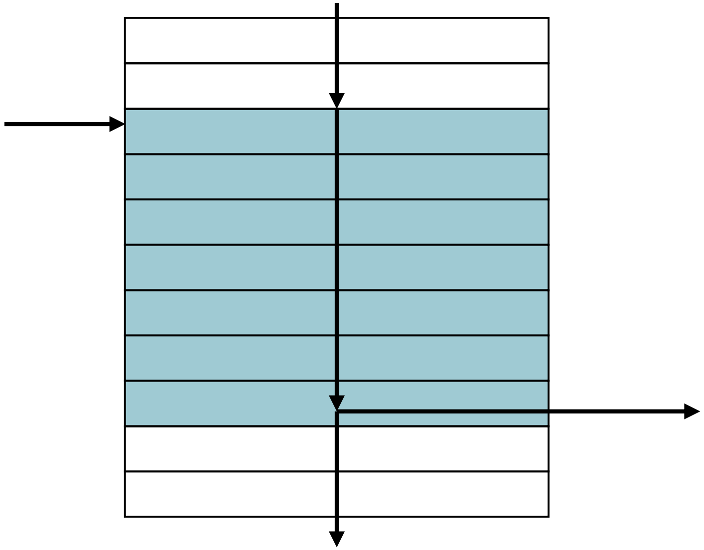
  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:



```
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
- A compilation breaks the program into basic blocks.

More Conditional Operations

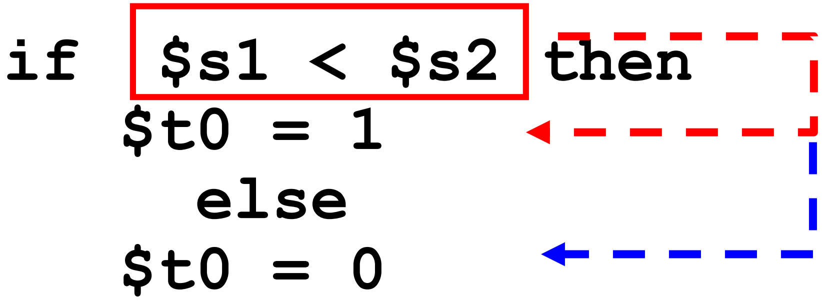
- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

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

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
-

```
slt $t0, $s1, $s2
```



```
if  $s1 < $s2 then
    $t0 = 1
      else
    $t0 = 0
```

Set on less than

The example code is not a real high level program code but explain algorithm

Control Flow

- Can use this instruction to build "`blt $s1, $s2, Label`"
  — can now build general control structures
  - **`blt, bgt, ble, bge:` pseudoinstructions**
  - **Availabe in assembly level but needs to translate into multiple real machine instructions**
  - **`E.g. blt -> slt and bne`**
  - **MIPS architecture does not include since it is complicated and may take longer clock cycles**
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

Branch Instruction Design

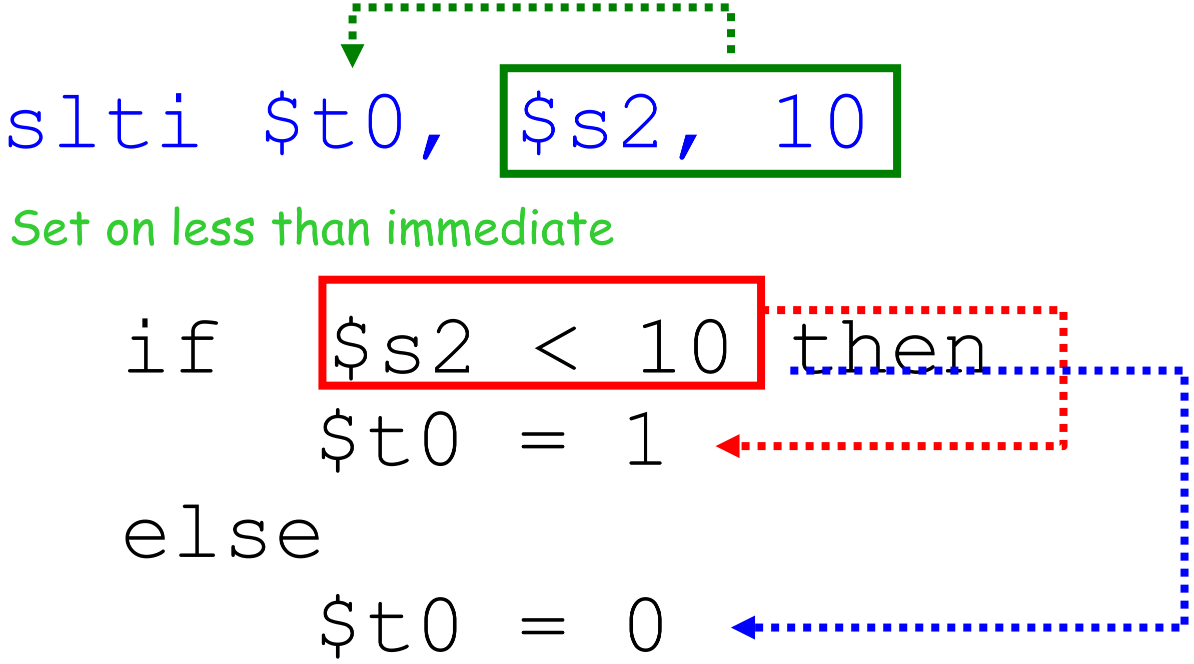
- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Here's the table for translating pseudoinstructions.

• bge $t0, $s0, LABEL	slt $at, $t0, $s0 beq $at, $zero, LABEL
• bgt $t0, $s0, LABEL	slt $at, $s0, $t0 bne $at, $zero, LABEL
• ble $t0, $s0, LABEL	slt $at, $s0, $t0 beq $at, $zero, LABEL
• blt $t0, $s0, LABEL	slt $at, $t0, $s0 bne $at, $zero, LABEL

Control Flow

- Constant comparison is popular



```
slti $t0, $s2, 10
```

Set on less than immediate

```
    if  $s2 < 10 then
        $t0 = 1
    else
        $t0 = 0
```

Signed vs. Unsigned

- 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 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Procedure

- In HL, procedures, sub modules or functions provides
  - *Easier reading and understanding*
  - *Allow code to be reused*
  - *Allow programmer to concentrate on just one portion of the task*
  - *Allow pass values and return results*
  - *** machine changes procedural steps to jump*

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

Register Usage

- $a0 – $a3: arguments (reg’s 4 – 7)
- $v0, $v1: result (return) 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)

Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result (return) 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)

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

Leaf Procedure Example

- 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

- C code:

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

$a0 : g

$a1 : h

$a2 : i

$a3 : j

$s0 : f

Leaf Procedure Example

- MIPS code:

```
leaf_example:
   addi $sp, $sp, -12
   sw   $t1, 8($sp)
   sw   $t0, 4($sp)
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   lw   $t0, 4($sp)
   lw   $t1, 8($sp)
   addi $sp, $sp, 12
   jr   $ra
```

Adjust stack to make room for 3 items

Procedure body

Result

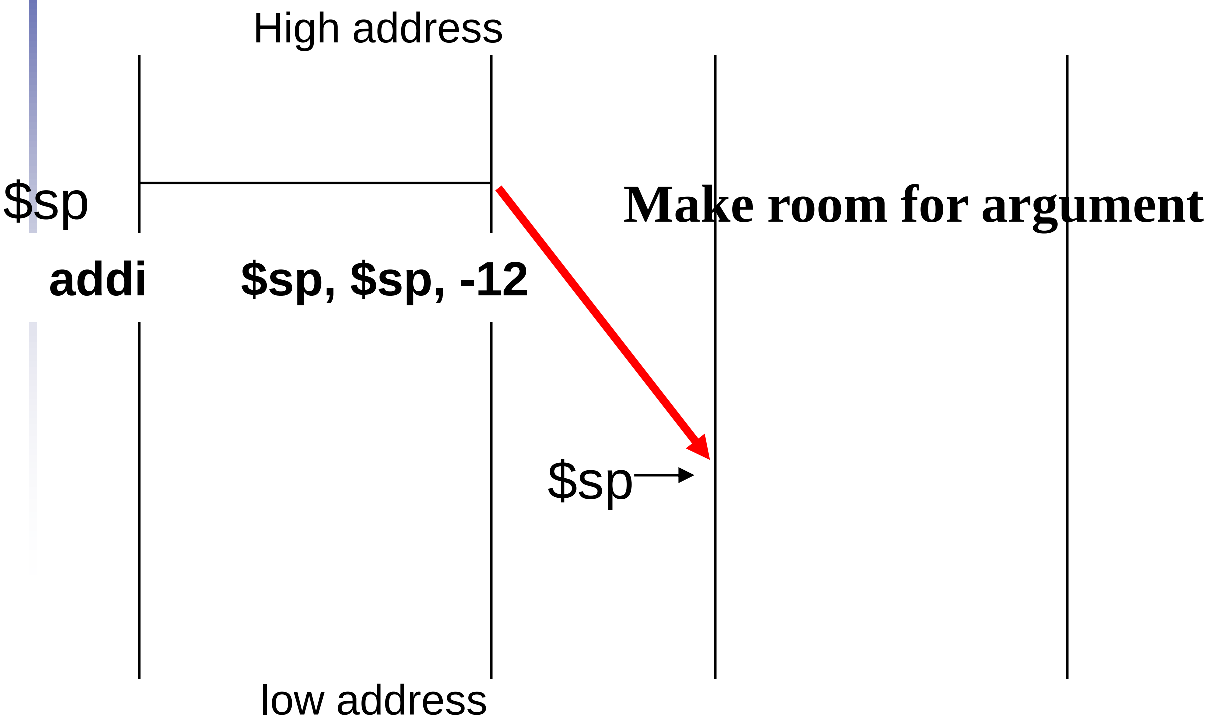
Restore $s0

Return

Stack Pointer

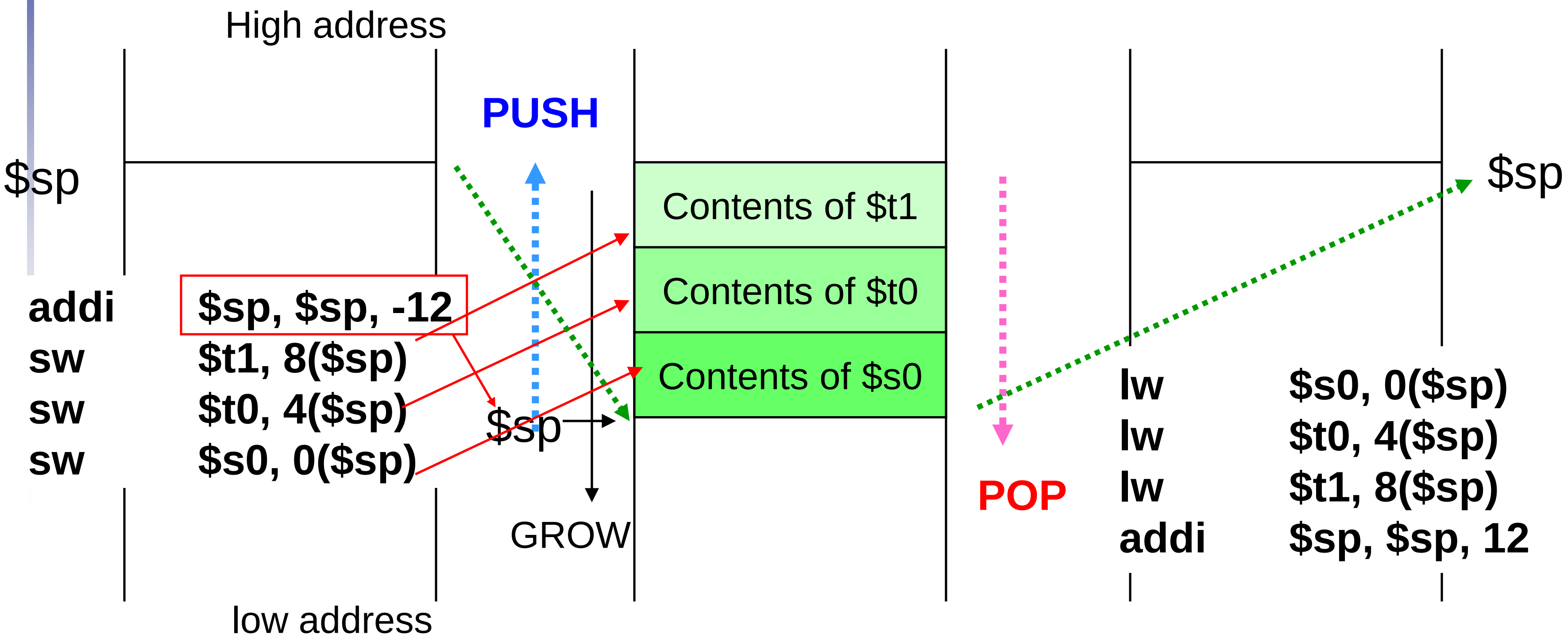
- Stack: LIFO data structure
- A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found.

Stack Pointer



addi $sp, $sp, -12

Stack Pointer



High address

PUSH

$sp

```
addi    $sp, $sp, -12
sw      $t1, 8($sp)
sw      $t0, 4($sp)
sw      $s0, 0($sp)
```

$sp

GROW

low address

Contents of $t1

Contents of $t0

Contents of $s0

POP

$sp

```
lw      $s0, 0($sp)
lw      $t0, 4($sp)
lw      $t1, 8($sp)
addi    $sp, $sp, 12
```

Leaf Procedure Example

- MIPS code:

```
leaf_example2:
    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
```

Using register spilling,
the caller does not expect
$t0 and $t1 to be preserved
across a procedure call

So simpler.

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

Non-Leaf Procedure Example

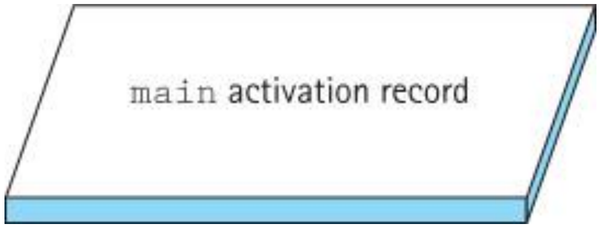
- C code:

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

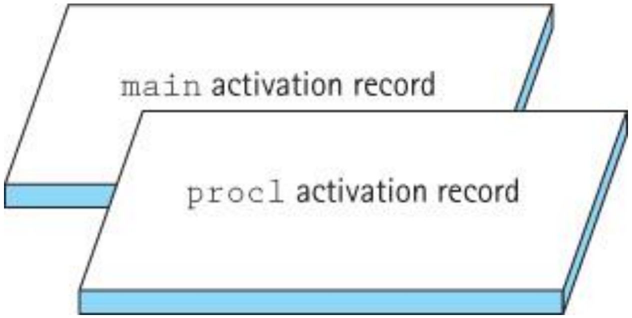
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

Consider a program whose main method calls proc1, which then calls proc2. When the program begins executing, the “main” activation record is generated:

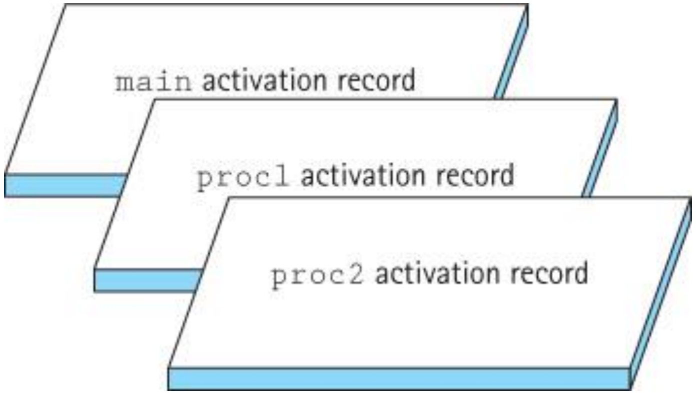


At the first method call, an activation record is generated for proc1:

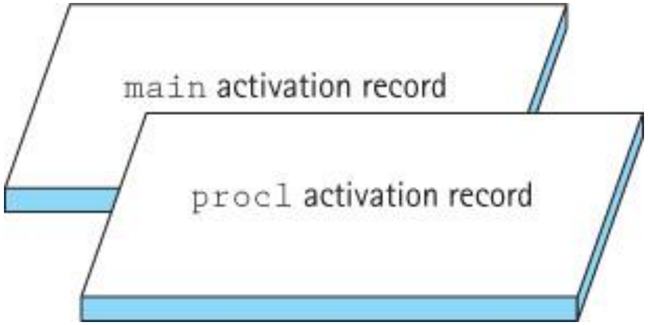


Non-Leaf Procedure Example

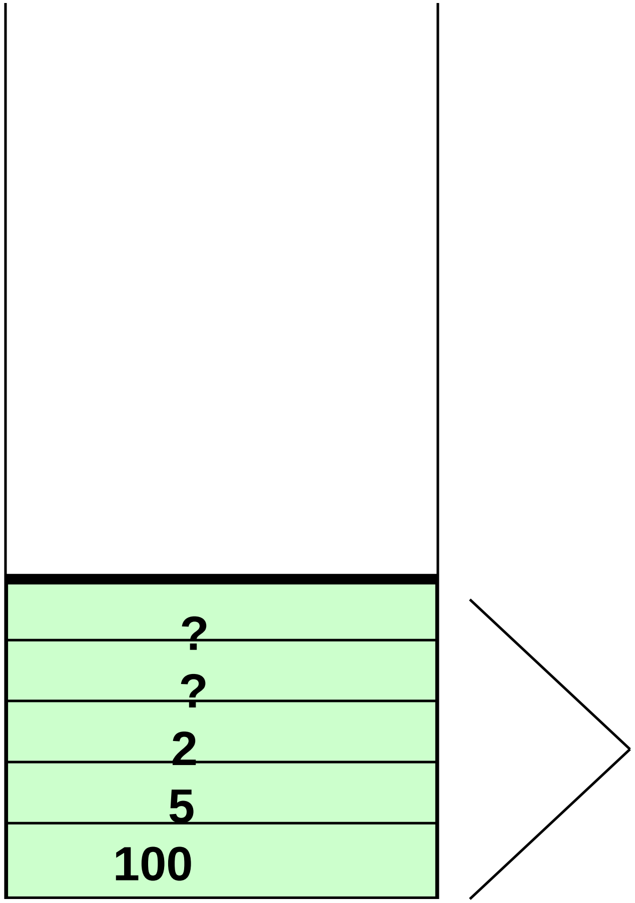
When proc2 is called from within proc1, its activation record is generated. Because proc1 has not finished executing, its activation record is still around:



When proc2 finishes executing, its activation record is released:



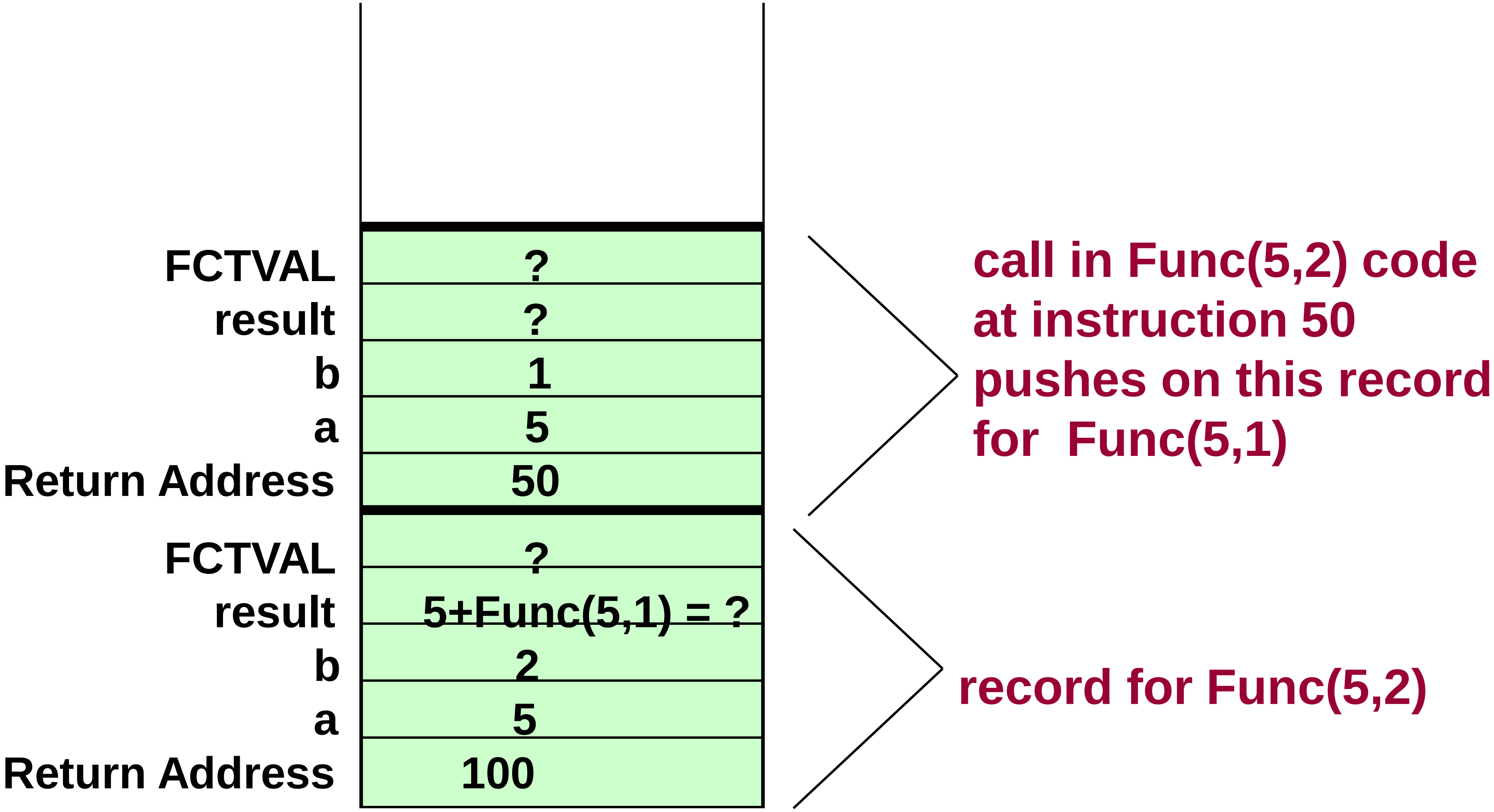
x = Func(5, 2);   // original call is instruction 100



FCTVAL	?
result	?
b	2
a	5
Return Address	100

original call
at instruction 100
pushes on this record
for Func(5,2)

```
x = Func(5, 2);          // original call at instruction 100
```

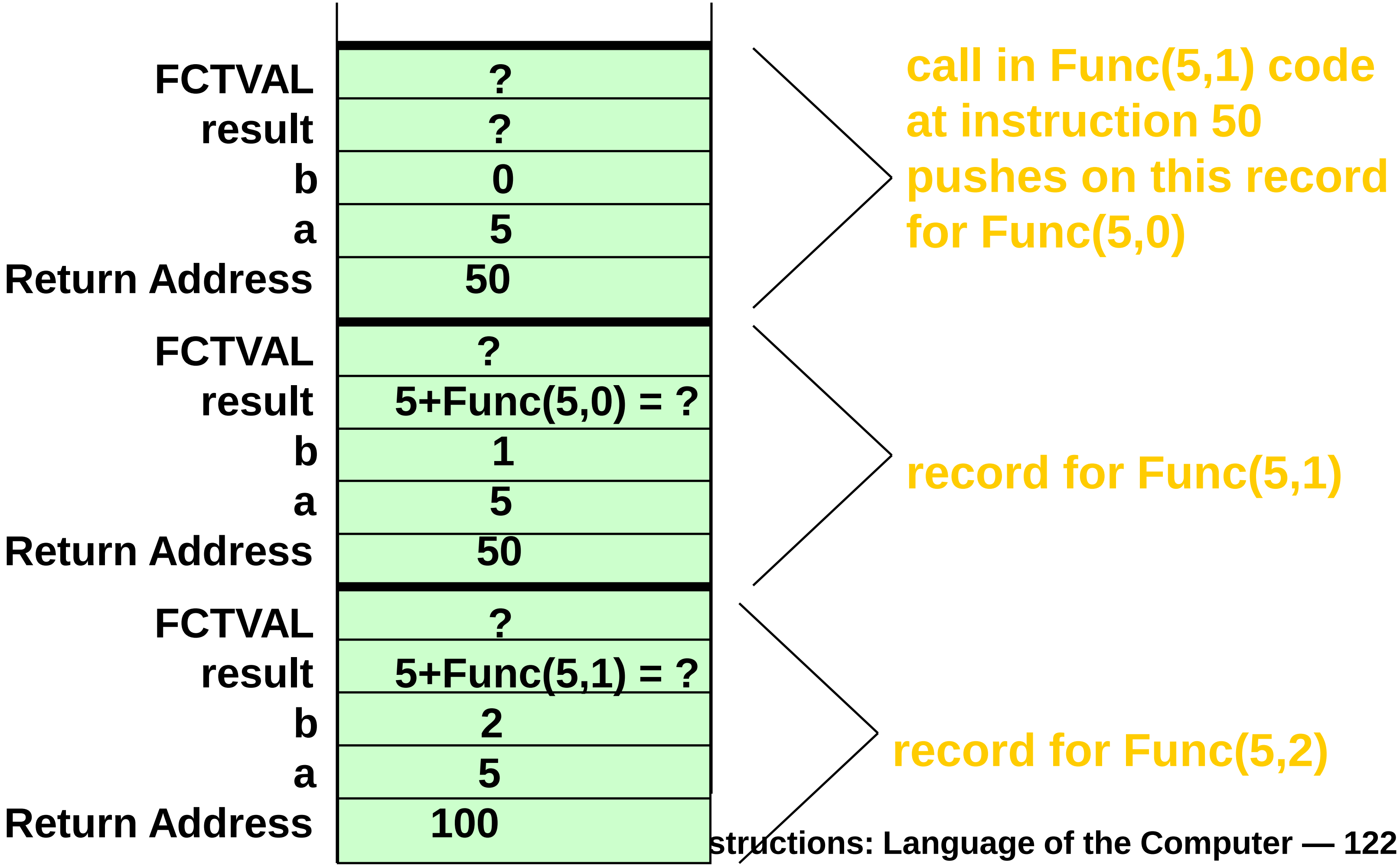


FCTVAL	?
result	?
b	1
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,1) = ?
b	2
a	5
Return Address	100

call in Func(5,2) code at instruction 50 pushes on this record for Func(5,1)

record for Func(5,2)

x = Func(5, 2);   // original call at instruction 100



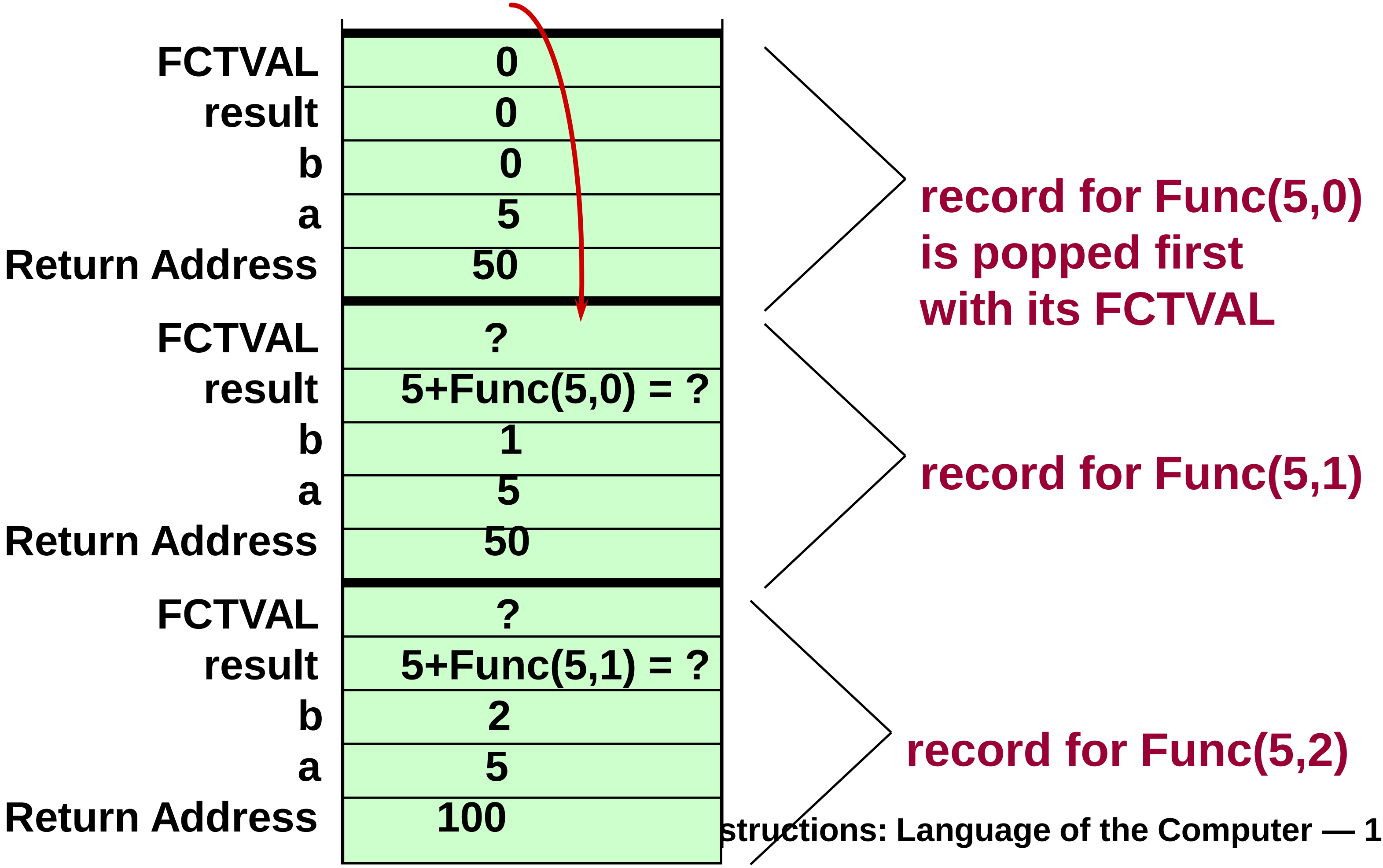
FCTVAL	?
result	?
b	0
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,0) = ?
b	1
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,1) = ?
b	2
a	5
Return Address	100

call in Func(5,1) code at instruction 50 pushes on this record for Func(5,0)

record for Func(5,1)

record for Func(5,2)

x = Func(5, 2); // original call at instruction 100



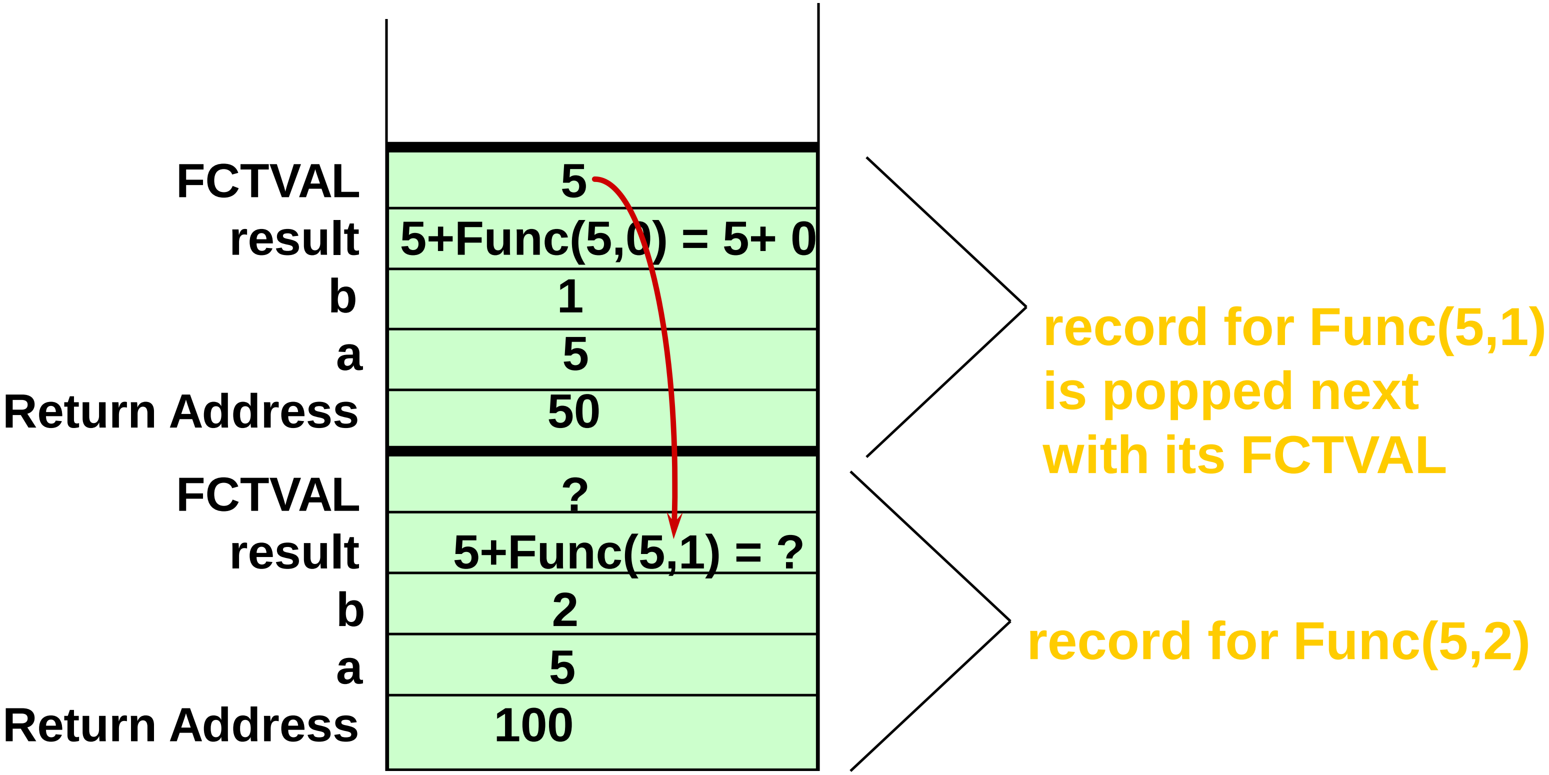
FCTVAL	0
result	0
b	0
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,0) = ?
b	1
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,1) = ?
b	2
a	5
Return Address	100

record for Func(5,0) is popped first with its FCTVAL

record for Func(5,1)

record for Func(5,2)

x = Func(5, 2);   // original call at instruction 100

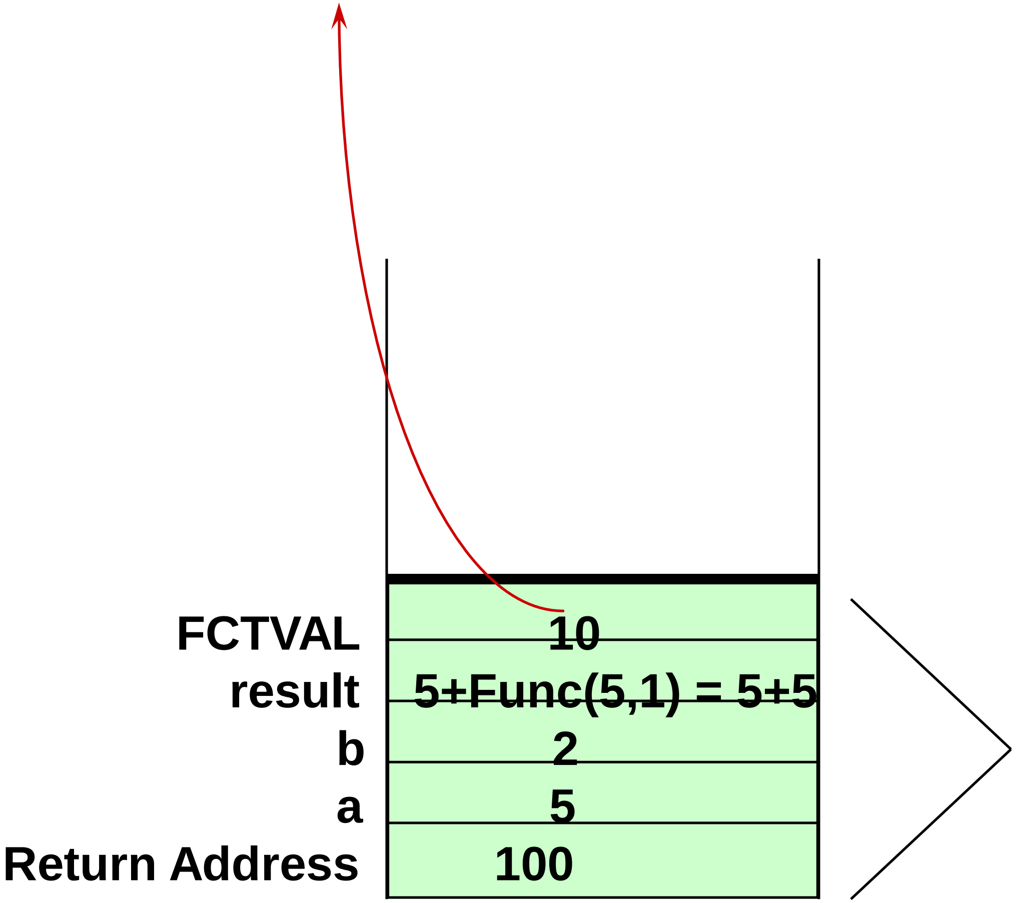


FCTVAL	5
result	5+Func(5,0) = 5+ 0
b	1
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,1) = ?
b	2
a	5
Return Address	100

record for Func(5,1) is popped next with its FCTVAL

record for Func(5,2)

x = Func(5, 2);   // original call at line 100



FCTVAL	10
result	5+Func(5,1) = 5+5
b	2
a	5
Return Address	100

record for Func(5,2)
is popped last
with its FCTVAL

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
```

Example. Nested procedure

```
fact:
    addi        $sp, $sp, -8  # adjust stack for 2 items: arg
                              and return address
    sw          $ra, 4($sp)   # save the return address
    sw          $a0, 0($sp)   # save the argument n
```

fact is called, sw saves an address in the program that called fact.

Parameter n : $a0 (argument register)

```
    slti        $t0, $a0, 1            # test for n < 1
    beq         $t0, $zero, L1         # if n >= 1, go to L1
```

Example. Nested procedure

If n < 1, returns 1 by adding 1 to 0

```
addi    $v0, $zero, 1      # return 1
addi    $sp, $sp, 8        # pop 2 items off stack
jr      $ra                # return to after jal
```

Example. Nested procedure

If n >= 1, the argument n is decreased, fact is called again

```
L1: addi $a0, $a0, -1   # n >= 1: argument gets (n – 1)
    jal       fact                  # call fact with (n – 1)
```

Where fact returns,

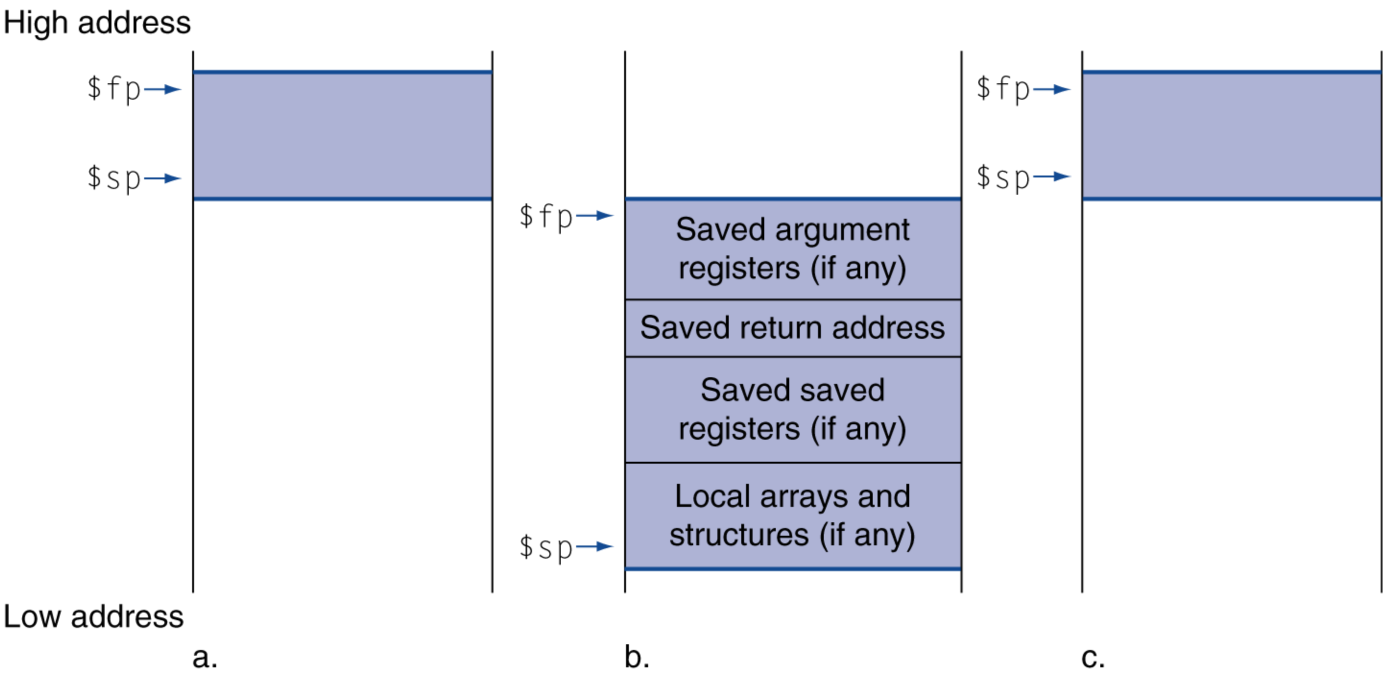
```
    lw   $a0, 0($sp)   # return from jal: restore argument n
    lw   $ra, 4($sp)   # restore the return address
    addi $sp, $sp, 8   # adjust stack pointer to pop 2 items
```

Example. Nested procedure

The value register $v0 gets the product of old argument $a0 and the current value of the value register.

```
mul $v0, $a0, $v0        # return n * fact (n – 1)
jr  $ra                  # return to the caller
```

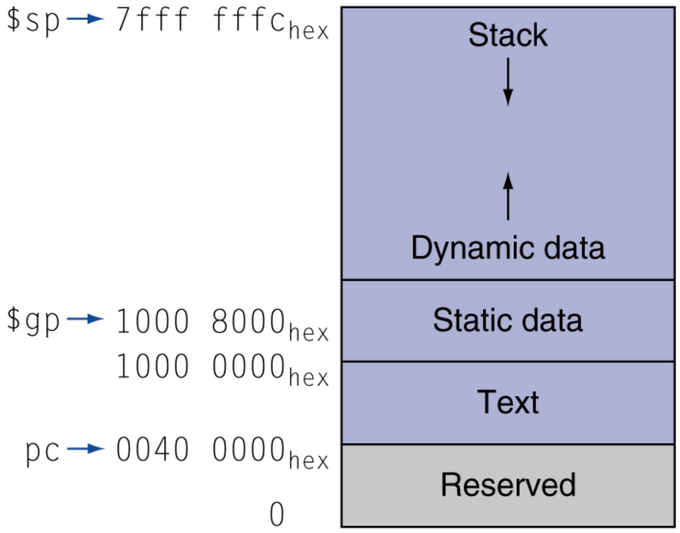
Local Data on the Stack


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

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

Memory Layout

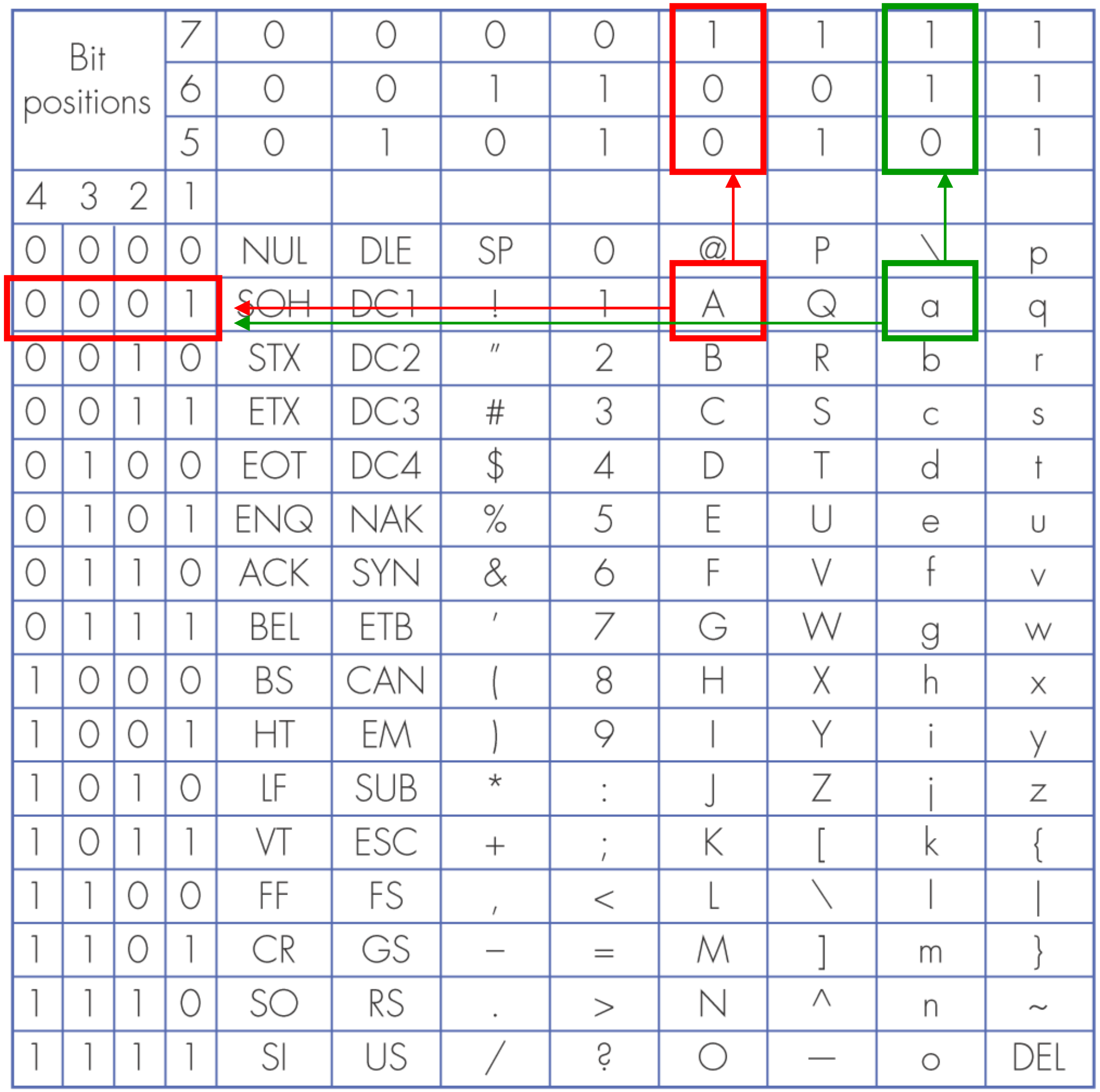
- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



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

ASCII



Bit positions			7	0	0	0	0	1	1	1	1
			6	0	0	1	1	0	0	1	1
			5	0	1	0	1	0	1	0	1
4	3	2	1								
0	0	0	0	NUL	DLE	SP	0	@	P	\	p
0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	STX	DC2	"	2	B	R	b	r
0	0	1	1	ETX	DC3	#	3	C	S	c	s
0	1	0	0	EOT	DC4	$	4	D	T	d	t
0	1	0	1	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	ACK	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	'	7	G	W	g	w
1	0	0	0	BS	CAN	(	8	H	X	h	x
1	0	0	1	HT	EM	)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
1	0	1	1	VT	ESC	+	;	K	[	k	{
1	1	0	0	FF	FS	,	<	L	\	l	|
1	1	0	1	CR	GS	–	=	M	]	m	}
1	1	1	0	SO	RS	.	>	N	^	n	~
1	1	1	1	SI	US	/	?	O	—	o	DEL

A: 65: 1000001

a: 97: 1100001

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

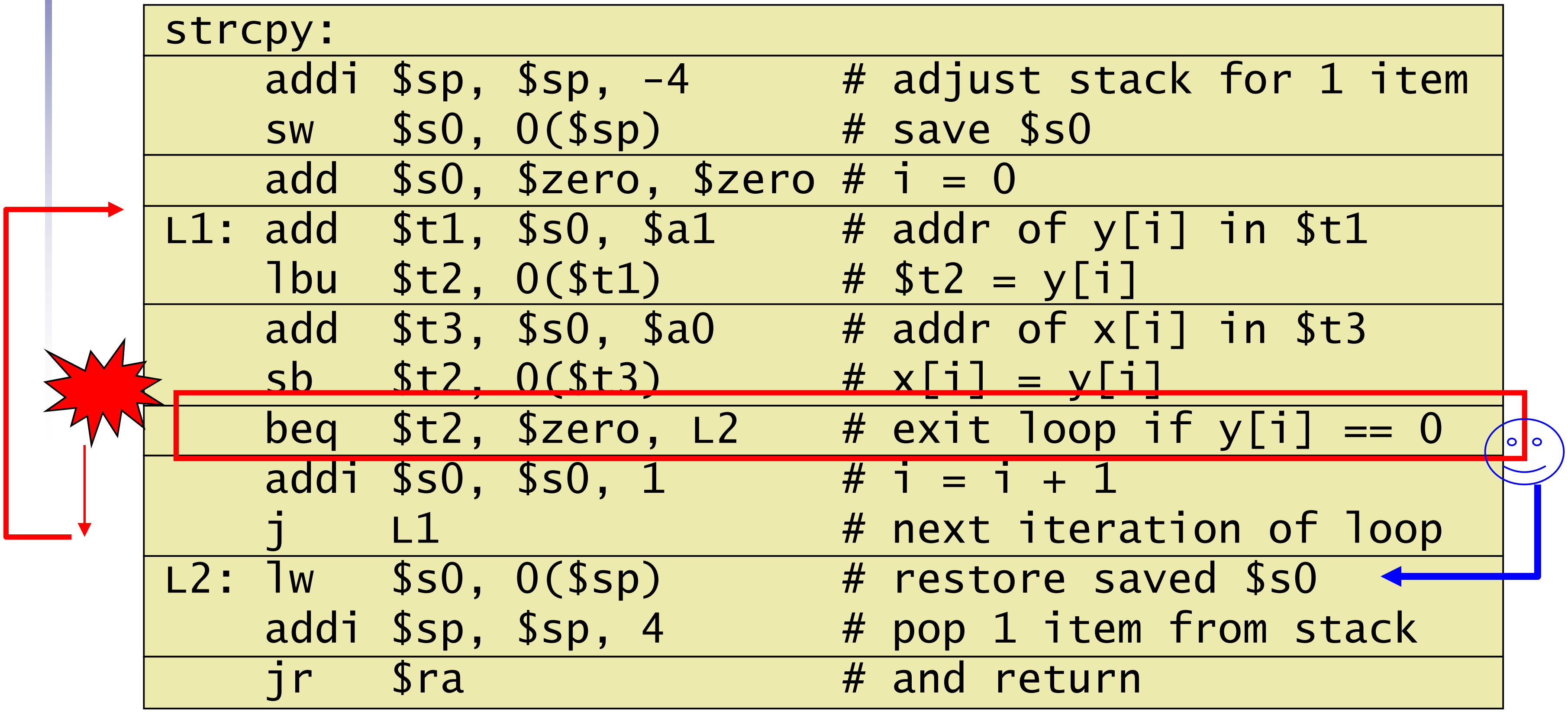
- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:



```
strcpy:
     addi $sp, $sp, -4        # adjust stack for 1 item
     sw   $s0, 0($sp)         # save $s0
     add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1        # addr of y[i] in $t1
     lbu  $t2, 0($t1)         # $t2 = y[i]
     add  $t3, $s0, $a0       # addr of x[i] in $t3
     sb   $t2, 0($t3)         # x[i] = y[i]
     beq  $t2, $zero, L2      # exit loop if y[i] == 0
     addi $s0, $s0, 1         # i = i + 1
     j    L1                  # next iteration of loop
L2: lw   $s0, 0($sp)          # restore saved $s0
     addi $sp, $sp, 4         # pop 1 item from stack
     jr   $ra                 # and return
```

32-bit Constants

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

```
lui rt, constant
```

Load upper immediate

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

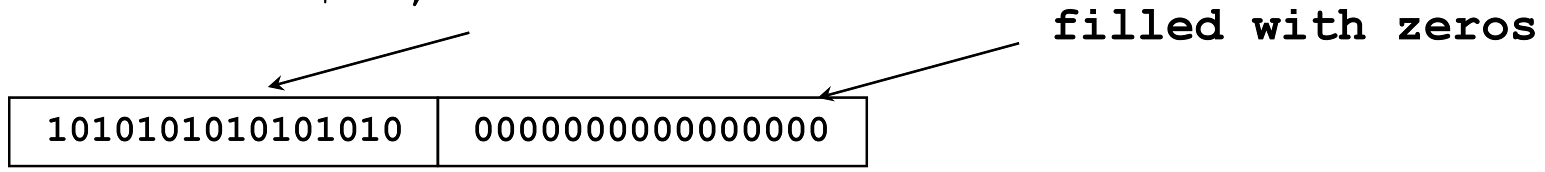
`lui $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

§2.10 MIPS Addressing for 32-Bit Immediates and Addresses

How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

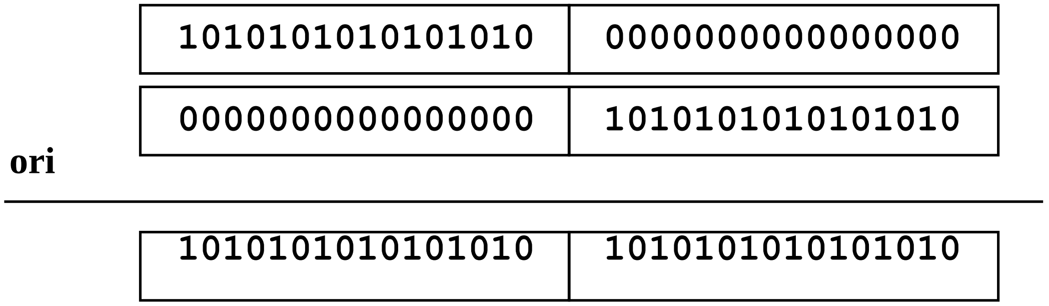


filled with zeros

1010101010101010	0000000000000000

- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



	1010101010101010	0000000000000000
ori	0000000000000000	1010101010101010
	1010101010101010	1010101010101010

How about larger constants?

- **How to add value of 4 million (4000000) into register $s0?**

  4000000 = 0000 0000 0011 1101 0000 1001 00000000

How about larger constants?

- **How to add value of 4 million (4000000) into register $s0?**

4000000 = 0000 0000 0011 1101 0000 1001 00000000

4000000 = 0000 0000 0011 1101 0000 1001 00000000

How about larger constants?

- How to add value of 4 million (4000000) into register $s0?

4000000 = 0000 0000 0011 1101 0000 1001 00000000

4000000 = 0000 0000 0011 1101 0000 1001 00000000

```
    lui $s0, 61          # decimal 61 = 0011 1101
                         # then $s0 becomes
#   0000 0000 0011 1101  0000 0000 0000 0000
```

How about larger constants?

- How to add value of 4 million (4000000) into register $s0?

4000000 = 0000 0000 0011 1101 0000 1001 0000 0000

4000000 = 0000 0000 0011 1101 0000 1001 0000 0000

```
lui $s0, 61          # decimal 61 = 0011 1101
# then $s0 becomes
#         0000 0000 0011 1101  0000 0000 0000 0000

ori $s0, $s0, 2304      # decimal 2304 = 0000 1001 0000 0000

# thus final value of $s0 becomes:
          0000 0000 0011 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

Program Counter

- 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 = $PC_{31\ldots28}$: (address × 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

Loop:	sll	$t1, $s3, 2	80000	0	0	19	9	4	0
	add	$t1, $t1, $s6	80004	0	9	22	9	0	32
	lw	$t0, 0($t1)	80008	35	9	8	0		
	bne	$t0, $s5, Exit	80012	5	8	21	2		
	addi	$s3, $s3, 1	80016	8	19	19	1		
	j	Loop	80020	2	20000				
Exit:	…		80024						

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

Label Exit is two instructions away from the branch instruction (80012).
It means PC contains address of current instruction plus offset address.
2 means 2 words or 8 bytes
So, 80016 + 8 bytes = 80024 bytes address

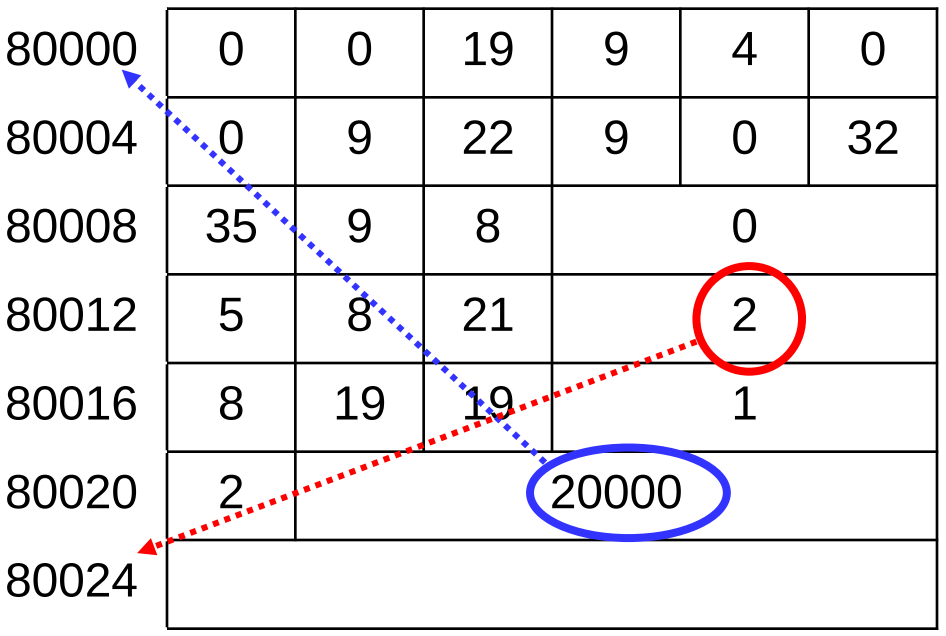
80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

What is the meaning of 20000?

20000 x 4 bytes = 80000 bytes of memory address



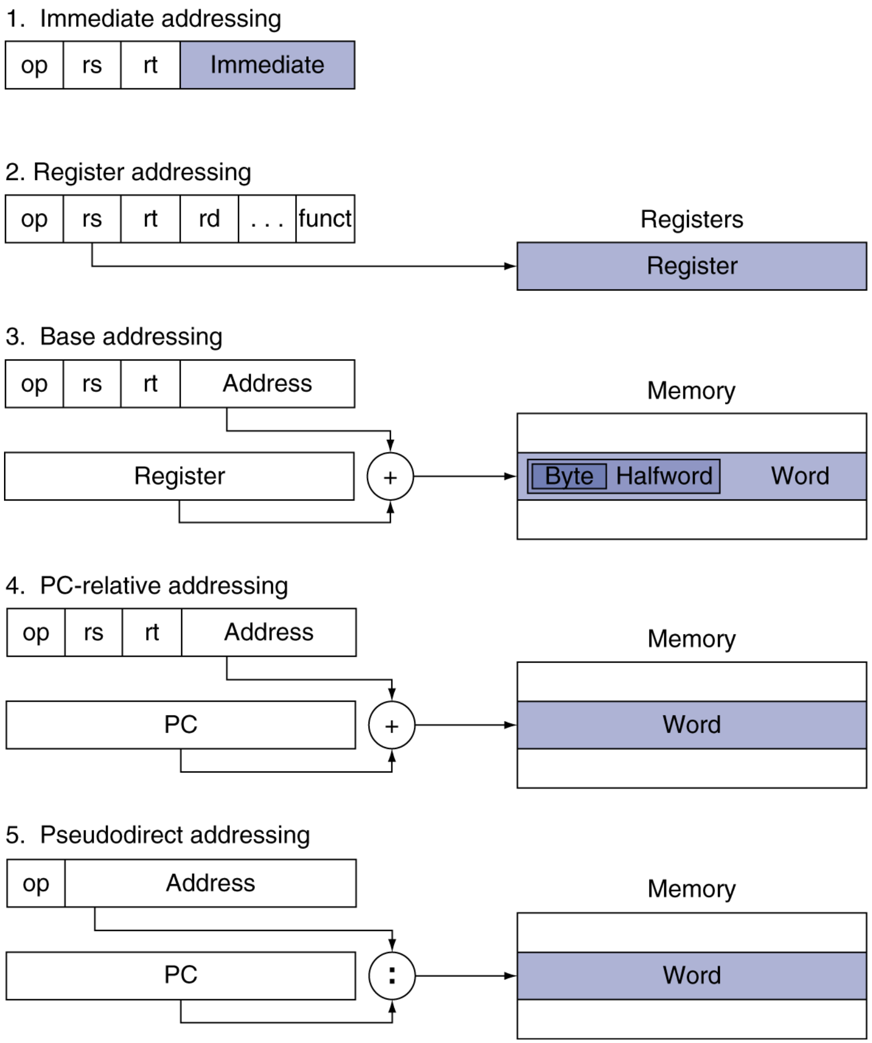
80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

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: …
```

Addressing Mode Summary



Decoding Machine Codes

- What is machine code of 00af8020 ?

Decoding Machine Codes

- What is machine code of 00af8020 ?
- First step: hexa -> binary conversion

00af8020 = 0000 0000 1010 1111 1000 0000 0010 0000

Decoding Machine Codes

- What is machine code of 00af8020 ?
- First step: hexa -> binary conversion

00af8020 = 0000 0000 1010 1111 1000 0000 0010 0000

Bit 31 ~ 29: 000

Bit 28 ~ 26: 000 thus it is R format instruction

Decoding Machine Codes

- What is machine code of 00af8020 ?
- First step: hexa -> binary conversion

00af8020 = 0000 0000 1010 1111 1000 0000 0010 0000

Bit 31 ~ 29: 000

Bit 28 ~ 26: 000 thus it is R format instruction

Restructure 32 bits to

op	rs	rt	rd	shamt	fuct
000000	00101	01111	10000	00000	100000

Bit 5 ~ 3: 100

Bit 2 ~ 0: 000 means add instruction..

Decoding Machine Codes

- What is machine code of 00af8020 ?
- First step: hexa -> binary conversion

00af8020 = 0000 0000 1010 1111 1000 0000 0010 0000

Bit 31 ~ 29: 000

Bit 28 ~ 26: 000 thus it is R format instruction

Restructure 32 bits to

```
op      rs    rt    rd    shamt  fuct
000000 00101 01111 10000 00000 100000
```

Bit 5 ~ 3: 100

Bit 2 ~ 0: 000 means add instruction..

Op: 0 rs: 5 rt: 15 rd: 16

```
add $s0, $a1, $t7
```

Synchronization

Danger of Data race

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

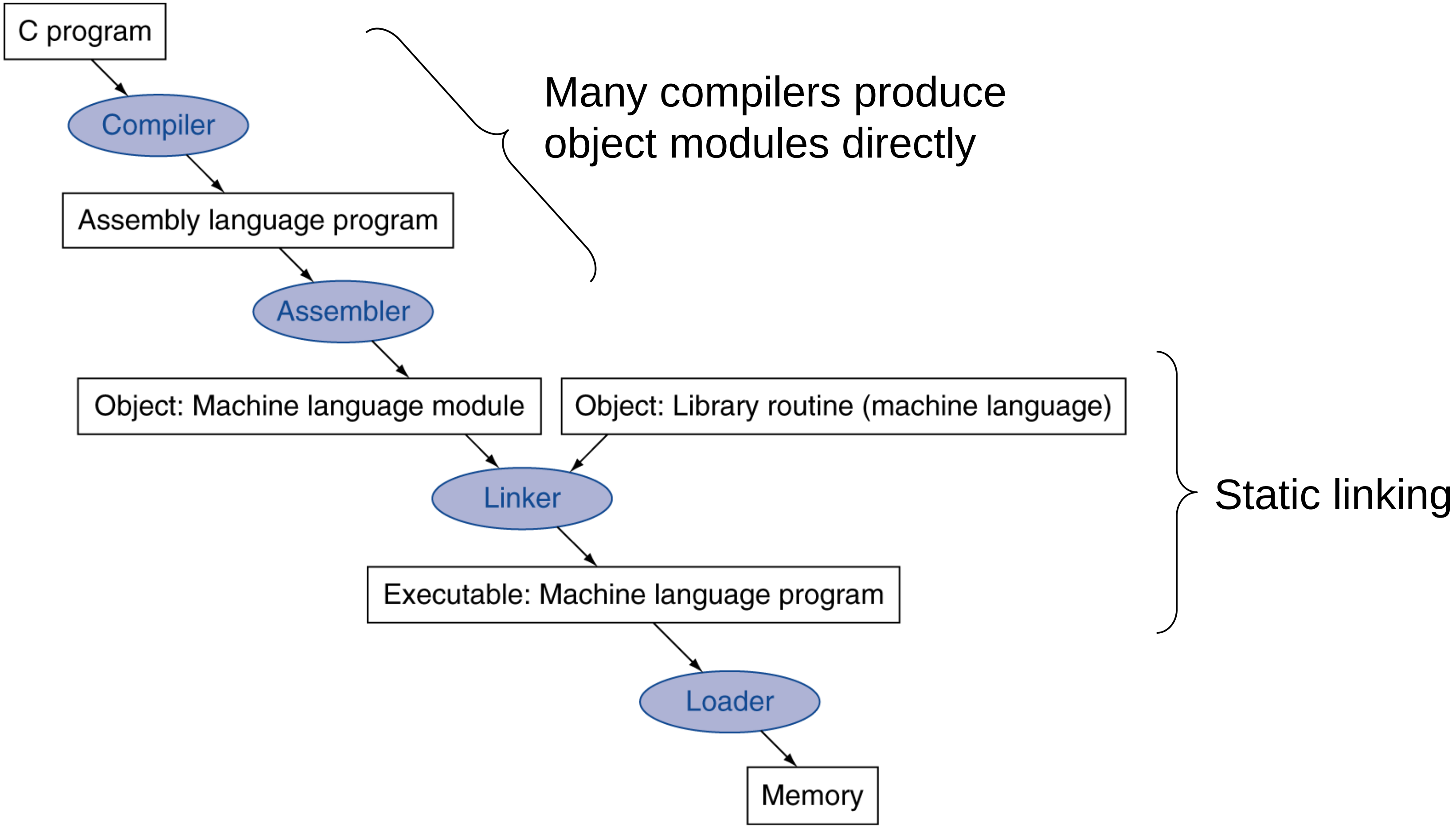
§2.11 Parallelism and Instructions: Synchronization

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Many compilers produce object modules directly

Static linking

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1    →  add $t0, $zero, $t1
blt $t0, $t1, L   → slt $at, $t0, $t1
                    bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

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

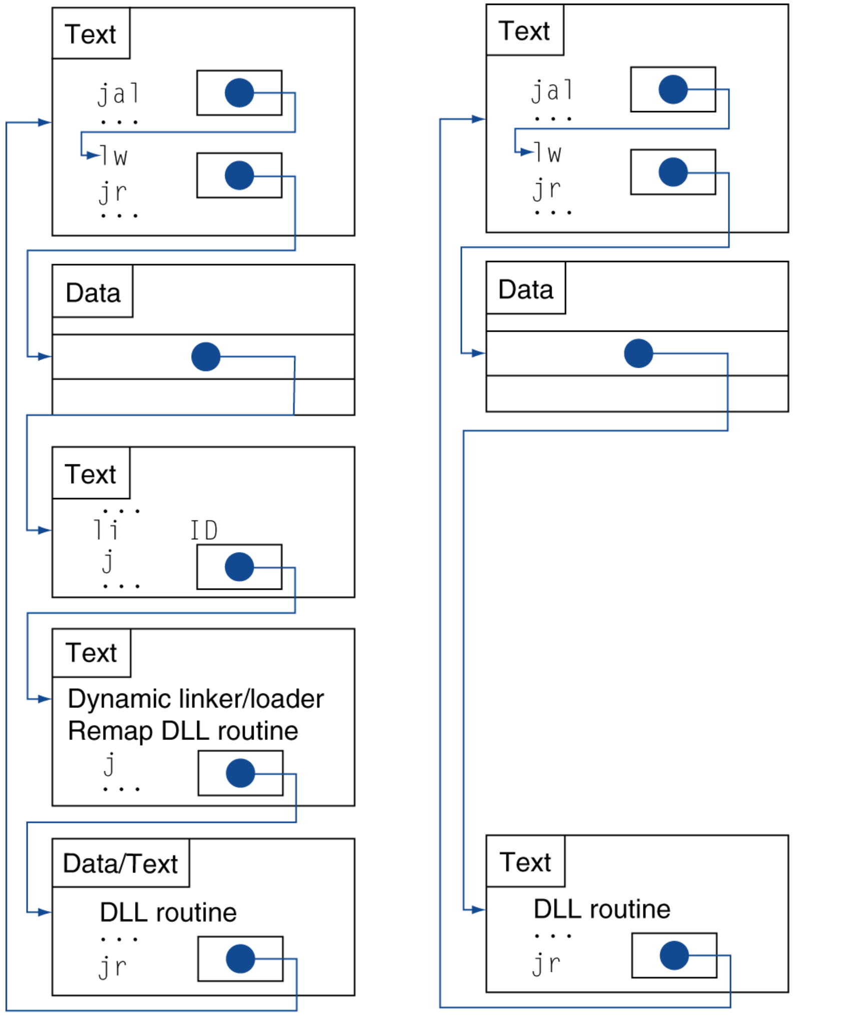
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

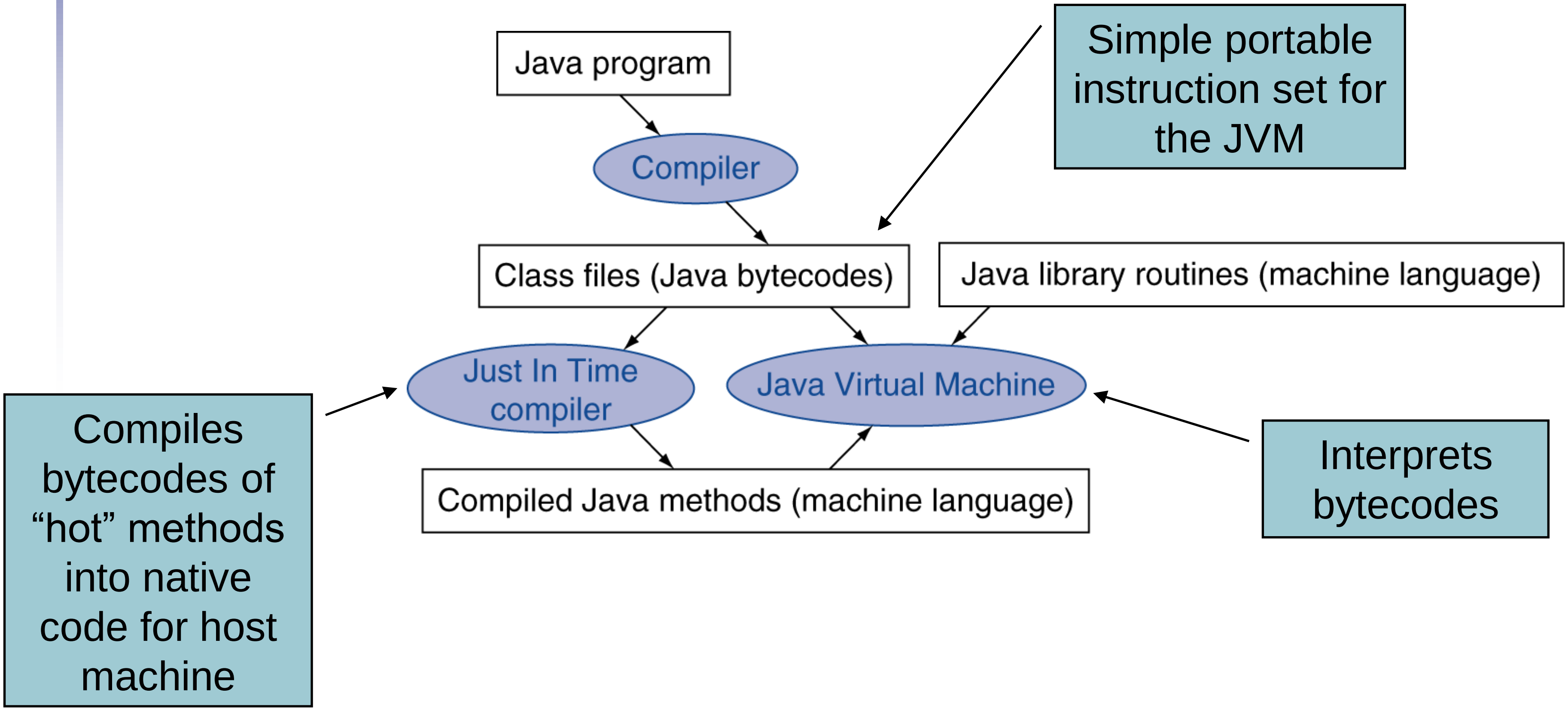
Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Starting Java Applications



Java program

Compiler

Class files (Java bytecodes)

Java library routines (machine language)

Just In Time compiler

Java Virtual Machine

Compiled Java methods (machine language)

Simple portable instruction set for the JVM

Compiles bytecodes of “hot” methods into native code for host machine

Interprets bytecodes

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

  - v in $a0, k in $a1, temp in $t0

§2.13 A C Sort Example to Put It All Together

The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
         move $s2, $a0            # save $a0 into $s2
         move $s3, $a1            # save $a1 into $s3
```

Move params

```
         move $s0, $zero          # i = 0
for1tst: slt  $t0, $s0, $s3       # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
```

Outer loop

```
         beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1        # j = i – 1
for2tst: slti $t0, $s1, 0         # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2   # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2         # $t1 = j * 4
         add  $t2, $s2, $t1       # $t2 = v + (j * 4)
         lw   $t3, 0($t2)         # $t3 = v[j]
         lw   $t4, 4($t2)         # $t4 = v[j + 1]
         slt  $t0, $t4, $t3       # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2   # go to exit2 if $t4 ≥ $t3
```

Inner loop

```
         move $a0, $s2            # 1st param of swap is v (old $a0)
         move $a1, $s1            # 2nd param of swap is j
         jal  swap                # call swap procedure
```

Pass params & call

```
         addi $s1, $s1, –1        # j –= 1
         j    for2tst             # jump to test of inner loop
```

Inner loop

```
exit2:   addi $s0, $s0, 1         # i += 1
         j    for1tst             # jump to test of outer loop
```

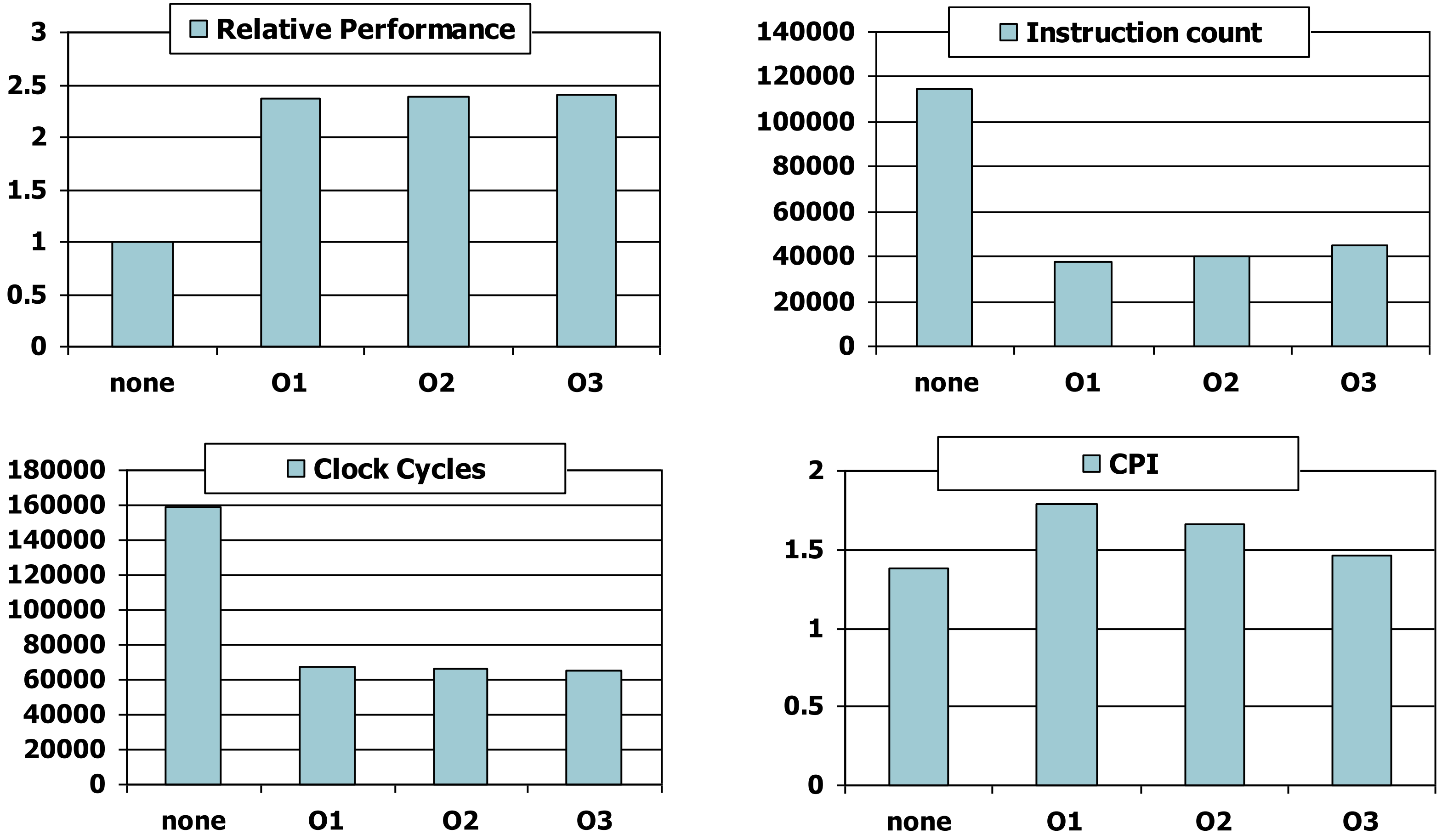
Outer loop

The Full Procedure

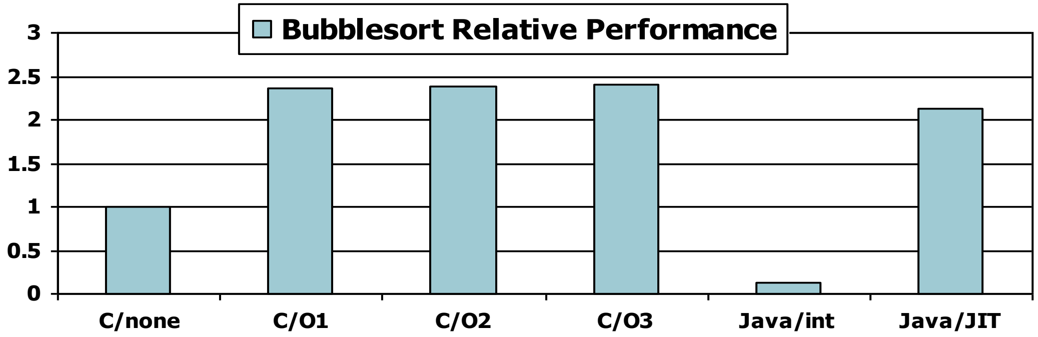
```
sort:    addi $sp,$sp, -20       # make room on stack for 5 registers
         sw $ra, 16($sp)         # save $ra on stack
         sw $s3,12($sp)          # save $s3 on stack
         sw $s2, 8($sp)          # save $s2 on stack
         sw $s1, 4($sp)          # save $s1 on stack
         sw $s0, 0($sp)          # save $s0 on stack
         …                       # procedure body
         …
         exit1: lw $s0, 0($sp)   # restore $s0 from stack
         lw $s1, 4($sp)          # restore $s1 from stack
         lw $s2, 8($sp)          # restore $s2 from stack
         lw $s3,12($sp)          # restore $s3 from stack
         lw $ra,16($sp)          # restore $ra from stack
         addi $sp,$sp, 20        # restore stack pointer
         jr $ra                  # return to calling routine
```

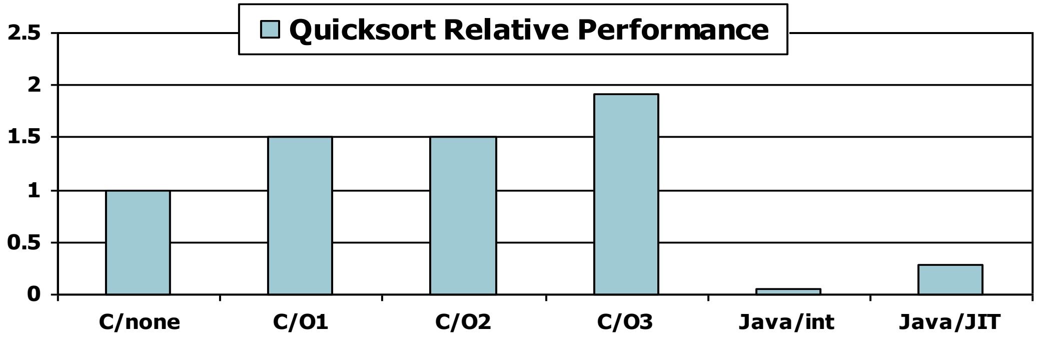
Appendix:Effect of Compiler Optimization

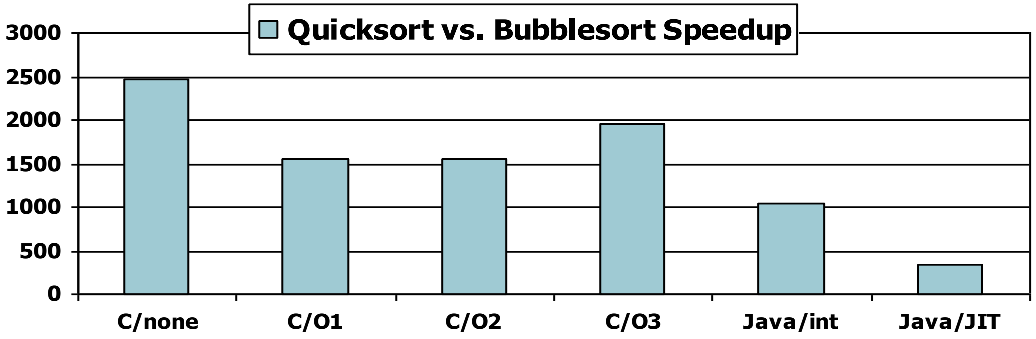
Compiled with gcc for Pentium 4 under Linux



Appendix: Effect of Language and Algorithm







Appendix: Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Appendix: Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Appendix: Example: Clearing and Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
       move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2     # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1    # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$a0      # p = & array[0]
       sll $t1,$a1,2     # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                         #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4    # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                         #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Appendix: Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Appendix: ARM & MIPS Similarities

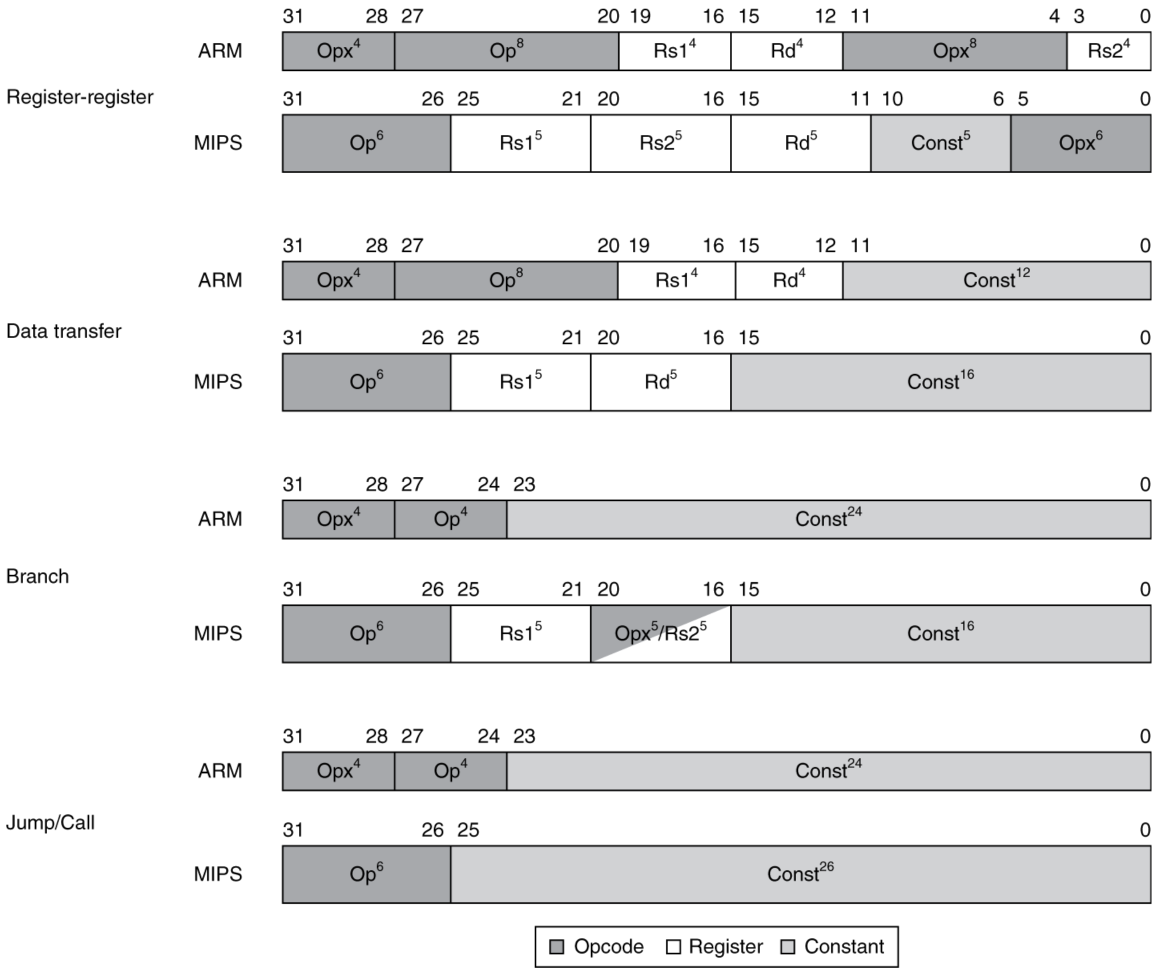
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	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

Appendix: Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Appendix: Instruction Encoding



Register-register

ARM

31 28	27 20	19 16	15 12	11 4	3 0
Opx^4	Op^8	Rs1^4	Rd^4	Opx^8	Rs2^4

MIPS

31 26	25 21	20 16	15 11	10 6	5 0
Op^6	Rs1^5	Rs2^5	Rd^5	Const^5	Opx^6

Data transfer

ARM

31 28	27 20	19 16	15 12	11 0
Opx^4	Op^8	Rs1^4	Rd^4	Const^{12}

MIPS

31 26	25 21	20 16	15 0
Op^6	Rs1^5	Rd^5	Const^{16}

Branch

ARM

31 28	27 24	23 0
Opx^4	Op^4	Const^{24}

MIPS

31 26	25 21	20 16	15 0
Op^6	Rs1^5	$\text{Opx}^5/\text{Rs2}^5$	Const^{16}

Jump/Call

ARM

31 28	27 24	23 0
Opx^4	Op^4	Const^{24}

MIPS

31 26	25 0
Op^6	Const^{26}

■ Opcode □ Register ▨ Constant

Appendix: 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

§2.17 Real Stuff: x86 Instructions

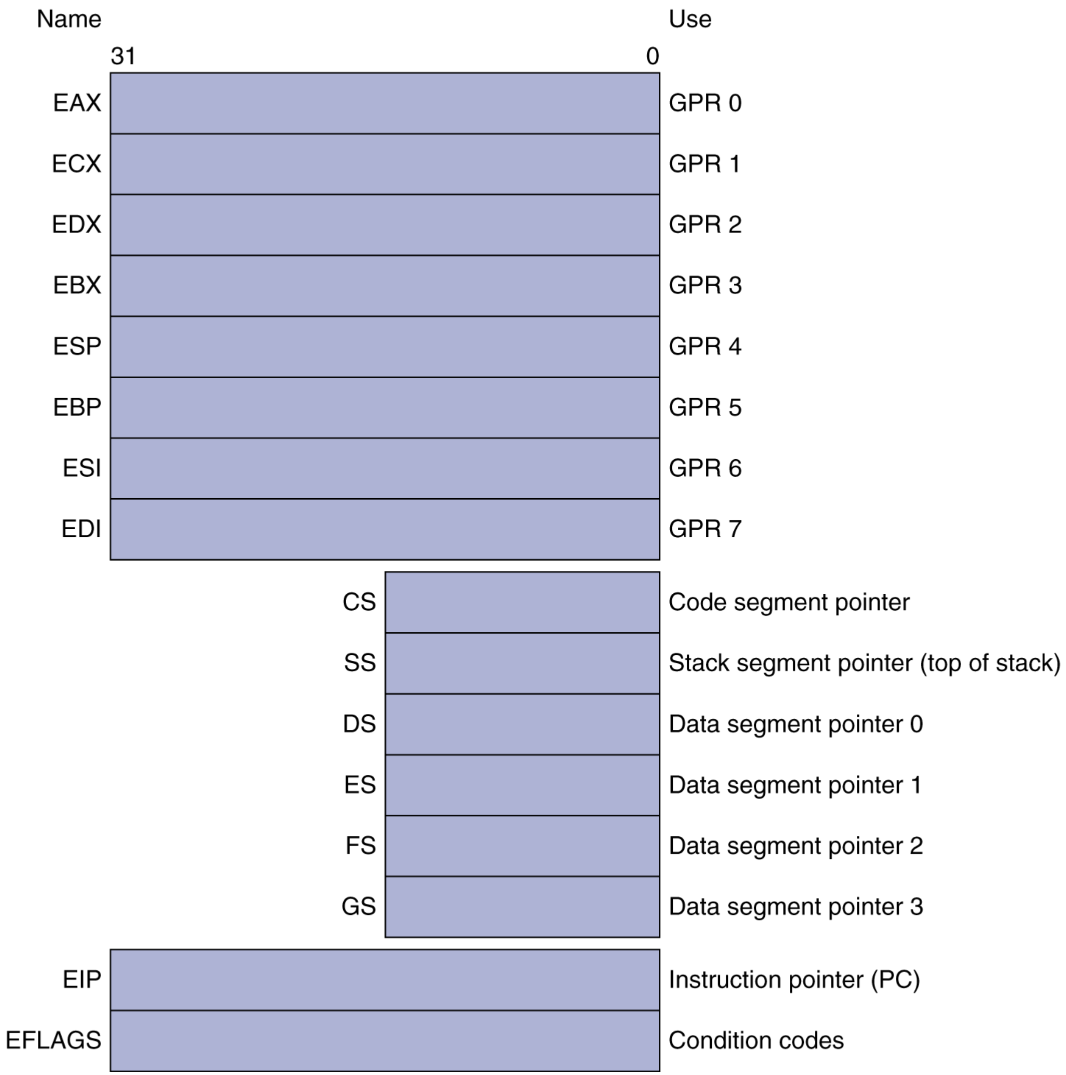
Appendix: The Intel x86 ISA

- 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

Appendix: 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

Appendix: Basic x86 Registers



31 0

Name	Use
EAX	GPR 0
ECX	GPR 1
EDX	GPR 2
EBX	GPR 3
ESP	GPR 4
EBP	GPR 5
ESI	GPR 6
EDI	GPR 7
CS	Code segment pointer
SS	Stack segment pointer (top of stack)
DS	Data segment pointer 0
ES	Data segment pointer 1
FS	Data segment pointer 2
GS	Data segment pointer 3
EIP	Instruction pointer (PC)
EFLAGS	Condition codes

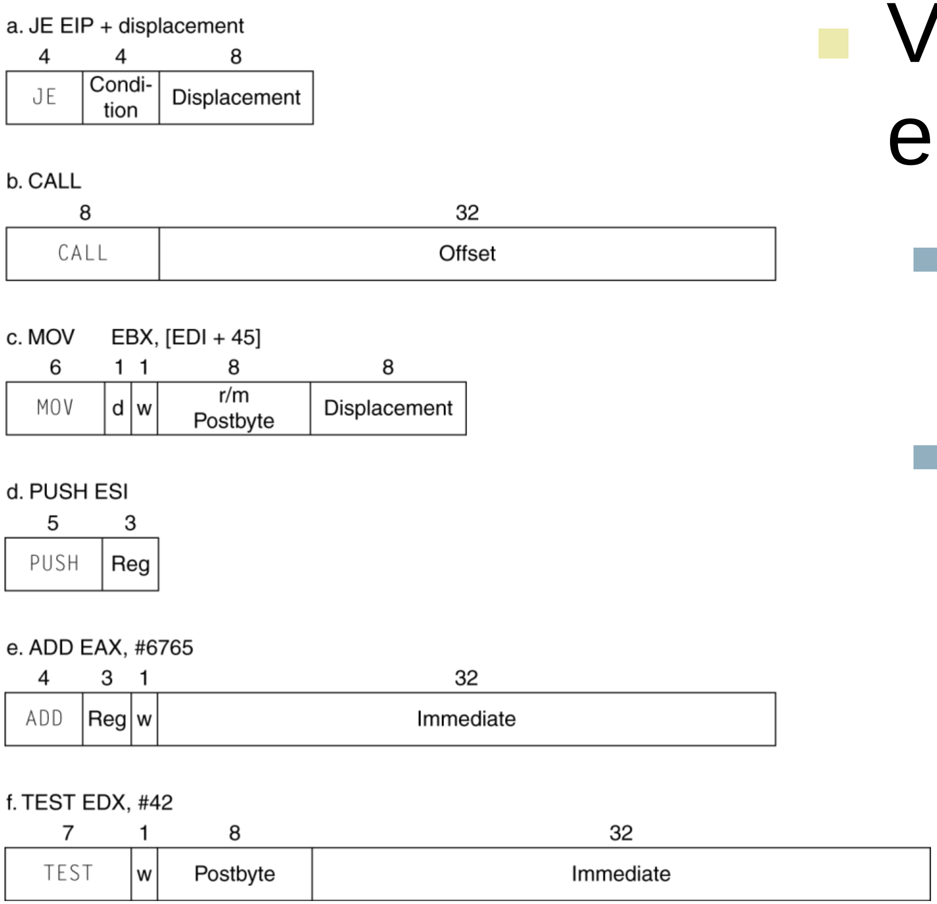
Appendix: Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

Appendix: x86 Instruction Encoding



a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Appendix: Implementing IA-32

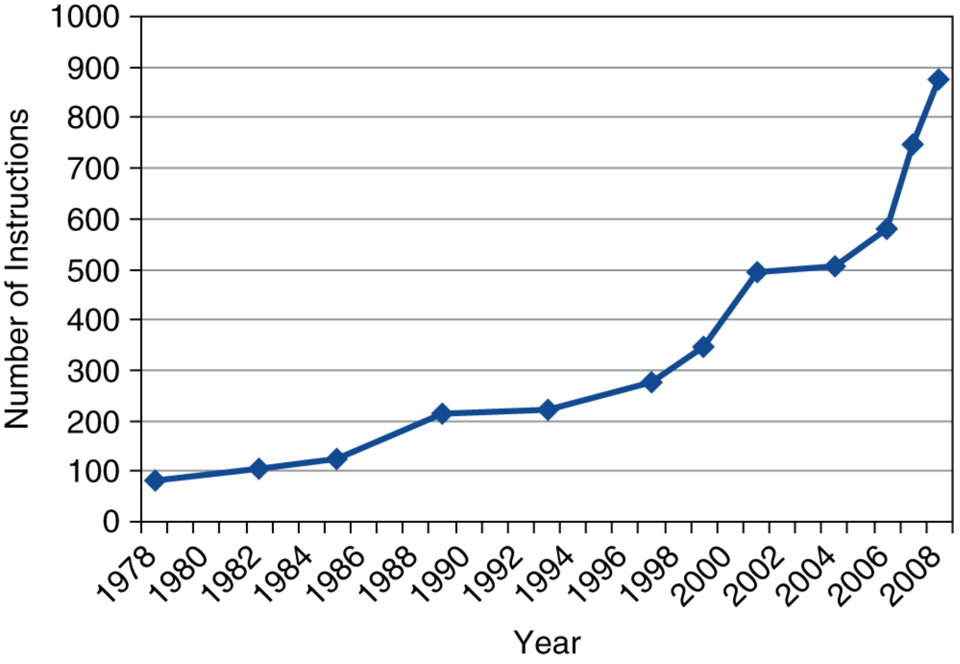
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
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

Appendix: 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

Appendix: Fallacies

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



x86 instruction set

Appendix: 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

Appendix: Concluding Remarks

§2.19 Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Appendix: 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%