Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Different processors have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - For simplified hardware implementation
- Many modern computers also have simple instruction sets
  - All have a common goal: ***to find a language that makes it easy to build the hardware***

Instruction Set Architecture, ISA

- A specification of a standardized programmer-visible interface to hardware, comprises of:
  - A set of instructions
    - instruction types
    - with associated argument fields, assembly syntax, and machine encoding.
  - A set of named storage locations
    - registers
    - memory
  - A set of addressing modes (ways to name locations)
  - Often an I/O interface
    - memory-mapped

High level language code : C, C++, Java, Fortan,

compiler

Assembly language code: architecture specific statements

assembler

Machine language code: architecture specific bit patterns

software

Instruction Set Architecture

hardware

ISA Design Issue

- Where are operands stored?
- How many explicit operands are there?
- How is the operand location specified?
- What type & size of operands are supported?
- What operations are supported?

Before answering these questions, let's consider more about

- Memory addressing
- Data operand
- Operations

Memory Addressing

- Most CPUs are byte-addressable and provide access for
  - Byte (8-bit)
  - Half word (16-bit)
  - Word (32-bit)
  - Double words (64-bit)

- How memory addresses are interpreted and how they are specified?
  - Little Endian or Big Endian
    - for ordering the bytes within a larger object within memory
  - Alignment or misaligned memory access
    - for accessing to an abject larger than a byte from memory
  - Addressing modes
    - for specifying constants, registers, and locations in memory

Byte-Order ("Endianness")

- **Little Endian**
  - The byte order put the byte whose address is "xx…x000" at the least-significant position in the double word
    - E.g. Intel, DEC, …
  - The bytes are numbered as

7	6	5	4	3	2	1	0
MSB							LSB

- **Big Endian**
  - The byte order put the byte whose address is "xx…x000" at the most-significant position in the double word
    - E.g. MIPS, IBM, Motorolla, Sun, HP, …
  - The byte address are numbered as

0	1	2	3	4	5	6	7
MSB							LSB

Little or Big Endian ?

- No absolute advantage for one over the other, but

  Byte order is a problem when exchanging data among computers

- Example
  - In C, **`int num = 0x12345678; // a 32-bit word`**,
  - how is `num` stored in memory?

	Big Endian
4n+3	**78**
4n+2	**56**
4n+1	**34**
4n+0	**12**

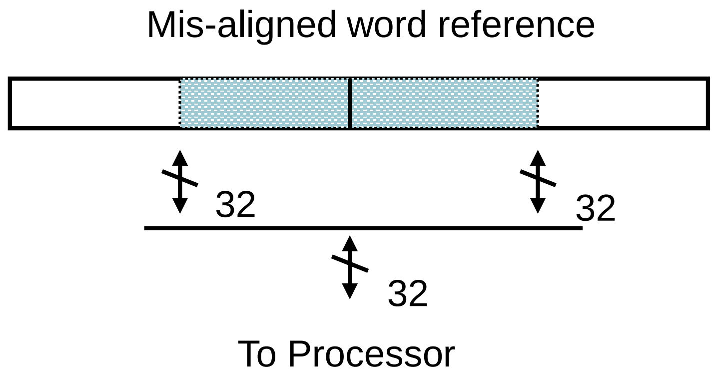
Big Endian

	Little Endian
4n+3	**12**
4n+2	**34**
4n+1	**56**
4n+0	**78**

Little Endian

Data Alignment

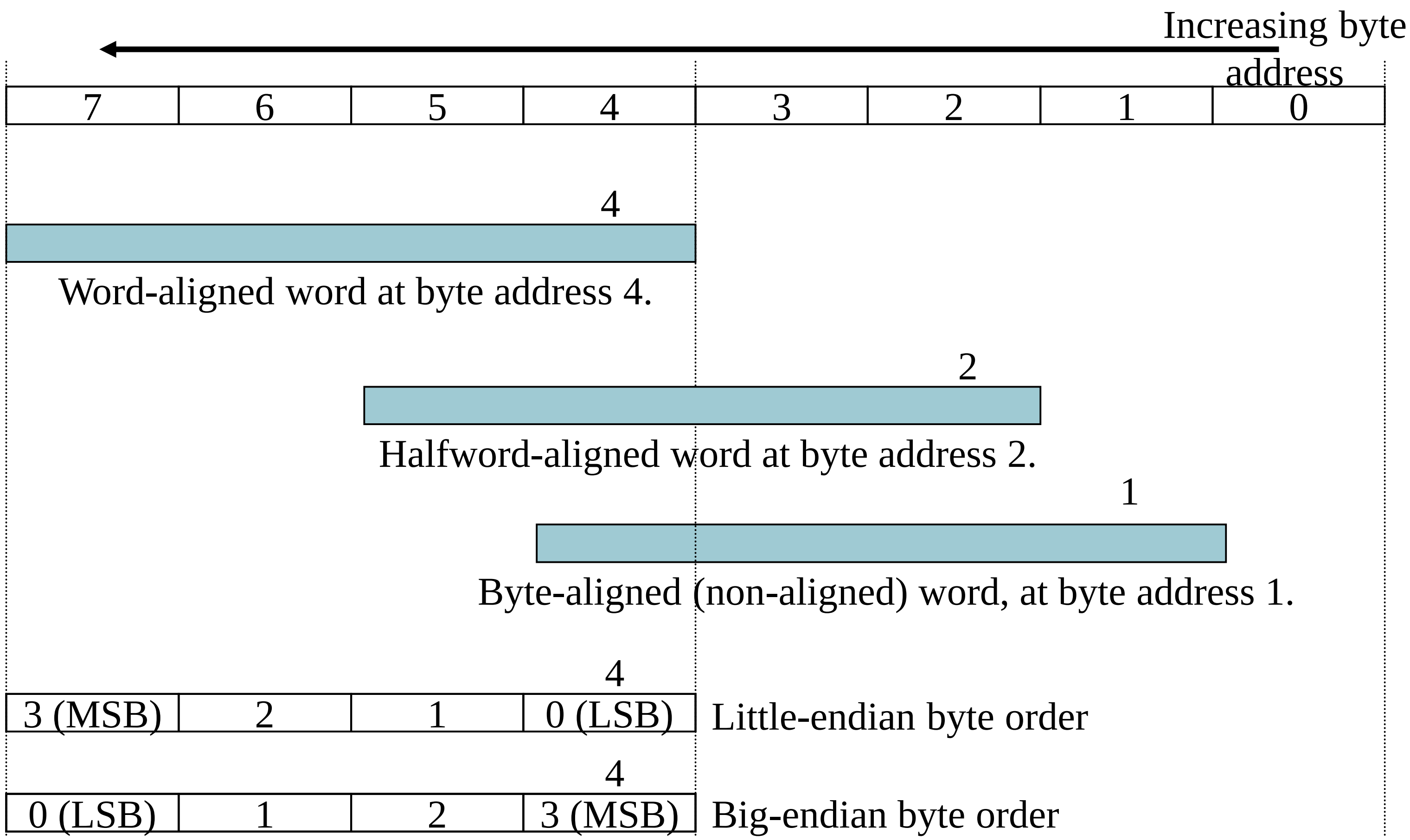
- The memory is typically aligned on a word or double-word boundary.
- An access to object of size S bytes at byte address A is called aligned if $A \bmod S = 0$.
- Access to an unaligned operand may require more memory accesses !!



Remarks

- **Unrestricted alignment access**
  - Software is simple
  - Hardware must detect misalignment and make more memory accesses
  - Expensive logic to perform detection
  - Can slow down all references
  - Sometimes required for backwards compatibility
- **Restricted alignment access**
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned

Summary: Endians & Alignment



Addressing Mode ?

- It answers the question:
  - Where can operands/results be located?

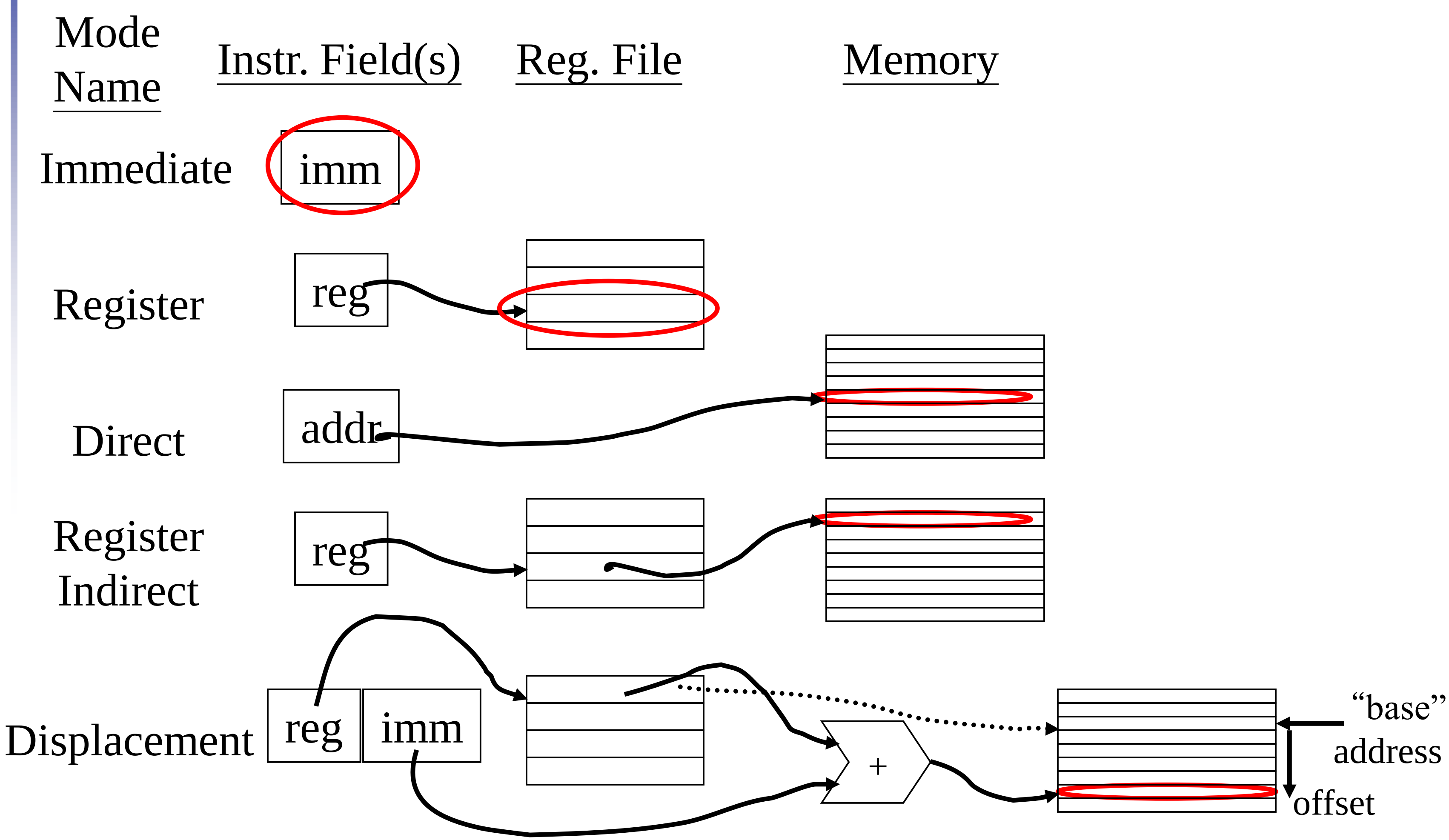
- Recall that we have two types of storage in computer : registers and memory
  - A single operand can come from either a register or a memory location
  - Addressing modes offer various ways of specifying the specific location

Addressing Mode Example

R: Register, M: Memory

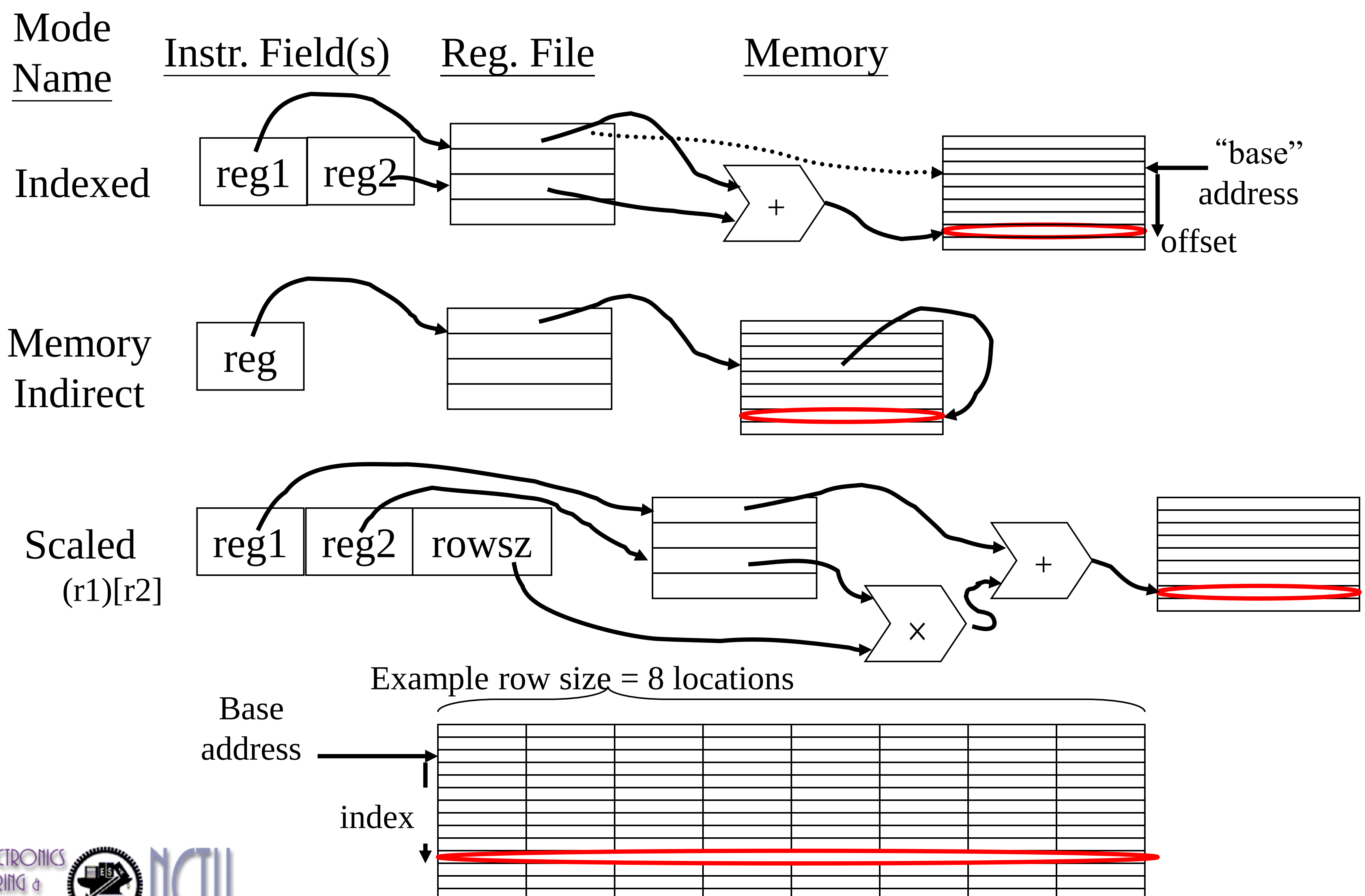
Addressing Mode	Example	Action
1. Register direct	`Add R1, R2, R3`	`R1 <- R2 + R3`
2. Immediate	`Add R1, R2, #3`	`R1 <- R2 + 3`
3. Register indirect	`Add R1, R2,(R3)`	`R1 <- R2 + M[R3]`
4. Displacement	`LD R1, 100(R2)`	`R1 <- M[100 + R2]`
5. Indexed	`LD R1, (R2 + R3)`	`R1 <- M[R2 + R3]`
6. Direct	`LD R1, (1000)`	`R1 <- M[1000]`
7. Memory Indirect	`Add R1, R2, @(R3)`	`R1 <- R2 + M[M[R3]]`
8. Auto-increment	`LD R1, (R2)+`	`R1 <- M[R2]` `R2 <- R2 + d`
9. Auto-decrement	`LD R1, (R2)-`	`R1 <- M[R2]` `R2 <- R2 - d`
10. Scaled	`LD R1, 100(R2)[R3]`	`R1 <- M[100+R2+R3*d]`

Addressing Modes Visualization (1)



all your base are belong to us

Addressing Modes Visualization (2)



How Many Addressing Mode ?

- A Tradeoff: complexity vs. instruction count
  - Should we add more modes?
    - Depends on the application class
    - Special addressing modes for DSP/GPU processors
      - Modulo or circular addressing
      - Bit reverse addressing
      - Stride, gather/scatter addressing
- Need to support at least three types of addressing mode
  - Displacement, immediate, and register indirect
  - They represent 75% -- 99% of the addressing modes in benchmarks
- The size of the address for displacement mode to be at least 12—16 bits (75% – 99%)
- The size of immediate field to be at least 8 – 16 bits (50%— 80%)
- DSPs rely on hand-coded libraries to exercise novel addressing modes

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS (since 1980s) commercialized by MIPS Technologies (www.mips.com)
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card and Appendix E
  - ARMv7 is similar to MIPS
  - Intel x86 is different from MIPS
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Arithmetic Operations

- Add/subtract, 3-operand instruction
  - Two sources and one destination

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

    - The words to the right of the sharp symbol (#) are comments for the human reader
- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled MIPS code:
  - break a C statement into several assembly instructions
  - introduce temporary variables

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

1. Register Operands

- Arithmetic instructions use register operands
  - Registers are primitives used in hardware design that are also visible to the programmer
- 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

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
```

operands are all registers !!

2. Memory Operands

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

Memory Operand Example 1

Access memory operand via addressing mode

- C code:

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

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

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

offset

base register

Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

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

Operand @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!

3. Immediate Operands or Constant

- 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
```

MIPS Registers

- 32 32-bit Registers with R0:=0
  - These registers are general purpose, any one can be used as an operand/result of an operation
  - But making different pieces of software work together is easier if certain conventions are followed concerning which registers are to be used for what purposes.
- Reserved registers: R1, R26, R27
  - R1 for assembler, R26-27 for OS
- Special usage:
  - R28: pointer register
  - R29: stack pointer
  - R30: frame pointer
  - R31: return address

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Policy of Use Conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Register 1 (`$at`) reserved for assembler, 26-27 for operating system

These conventions are usually suggested by the vendor and supported by the compilers

Binary Representation of Integers

- Number can be represented in any base
- Hexadecimal/Binary/Decimal representations

  $\mathtt{ACE7_{hex}} = \mathtt{1010\ 1100\ 1110\ 0111_{bin}} = \mathtt{44263_{dec}}$

  - most significant bit, MSB, usually the leftmost bit
  - least significant bit, LSB, usually the rightmost bit
- Ideally, we can represent any integer if the bit width is unlimited
- Practically, the bit width is limited and finite…
  - for a 8-bit byte → 0~255 ($0 \sim 2^8 - 1$)
  - for a 16-bit halfword → 0~65,535 ($0 \sim 2^{16} - 1$)
  - for a 32-bit word → 0~4,294,967,295 ($0 \sim 2^{32} - 1$)

Unsigned Binary Integers

- Given an n-bit number

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

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

Signed Integers or Numbers

- Unsigned number is mandatory
  - Eg. Memory access, PC, SP, RA
- Sometimes, negative integers are required in arithmetic operation
  - a representation that can present both positive and negative integers is demanded
- 3 well-known methods for signed integers
  - Sign and magnitude
  - 1's complement
  - 2's complement

Sign and Magnitude Representation

- Use the MSB as the **sign** bit
  - 0 for positive and 1 for negative
- If the bit width is n
  - range ➔ $-(2^{n-1} - 1) \sim 2^{n-1} - 1$; **$2^n - 1$ different numbers**
  - e.g., for a byte ➔ –127 ~ 127
- Examples
  - 00000110 ➔ +6
  - 10000111 ➔ –7
- Shortcomings
  - 2 0's; positive 0 and negative 0; 00000000 and 10000000
  - relatively complicated HW design (e.g., adder)

1's Complement Representation

+7 ➔ 0000 0111

–7 ➔ 1111 1000 (bit inverting)

- If the bit width is n
  - range ➔ $-(2^{n-1} - 1) \sim 2^{n-1} - 1$; **$2^n - 1$ different numbers**
  - e.g., for a byte ➔ –127 ~ 127
- The MSB implicitly serves as the sign bit
  - except for –0
- Shortcomings
  - 2 0's; positive 0 and negative 0; 00000000 and 11111111
  - relatively complicated HW design (e.g., adder)

2's Complement Representation

+7 ➔ 0000 0111

–7 ➔ 1111 1001 (bit inverting first then add 1)

- The MSB implicitly serves as the sign bit
- 2's complement of 10000000 ➔ 10000000
  - this number is defined as –128
- If the bit width is n
  - range ➔ $\mathbf{-2^{n-1}} \sim 2^{n-1} - 1$; **$2^n$ different numbers**
  - e.g., for a byte ➔ **–128** ~ 127
- Relatively easy hardware design

- Virtually, all computers use 2's complement representation

2's-Complement Signed Integers (1/2)

- Given an n-bit number

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

- Range: $-2^{n-1} \sim +2^{n-1} - 1$
- 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
  - $-2{,}147{,}483{,}648 \sim +2{,}147{,}483{,}647$

2’s-Complement Signed Integers (2/2)

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

Signed Negation

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

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

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

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

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

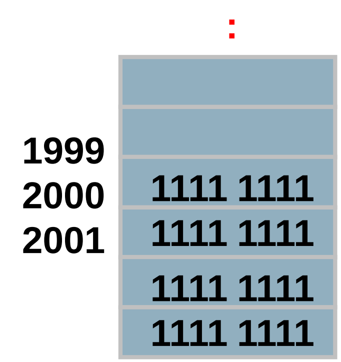
Example : lbu vs lb

- We want to load a BYTE into **`$s3`** from the address 2000

  After the load, what is the value of **`$s3`** ?

  - A1: 0000 0000 0000 0000 0000 0000 1111 1111 (255) ?
  - A2: 1111 1111 1111 1111 1111 1111 1111 1111 (–1) ?

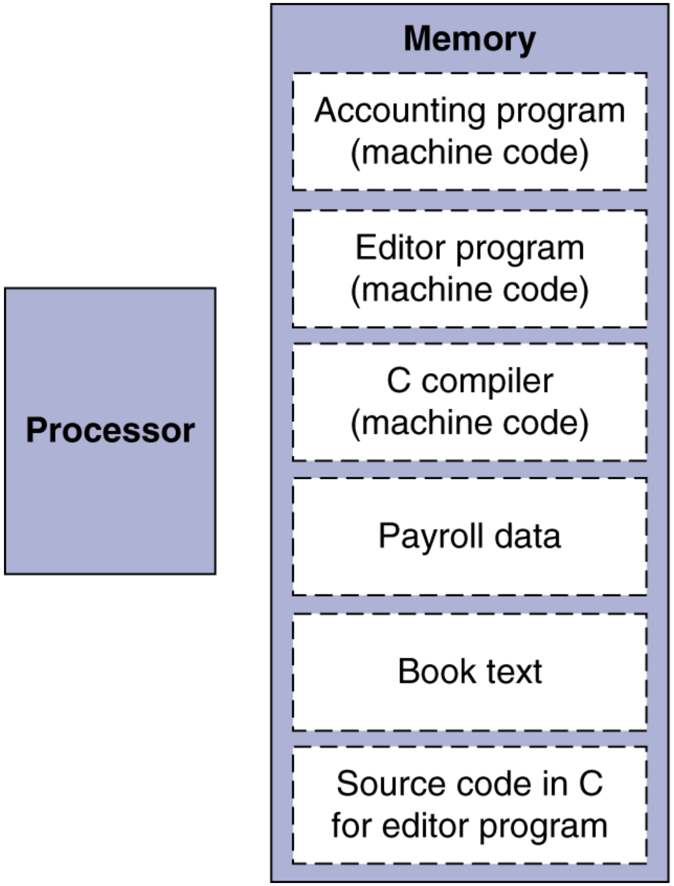
- Signed (A2) ➔ **`lb $s3, 0($s0)`**
- Unsigned (A1) ➔ **`lbu $s3, 0($s0)`**



Assume $s0 = 2000

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

Representing Instructions

- Instructions are encoded in binary
  - Called (binary) machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity !!
- Register numbers (5-bit representation)
  - \$t0 – \$t7 are reg's 8 – 15
  - \$t8 – \$t9 are reg's 24 – 25
  - \$s0 – \$s7 are reg's 16 – 23

MIPS R-format Instructions

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

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

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

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

special	\$s1	\$s2	\$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

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

Hexadecimal

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

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

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

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

Concluding Remarks

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

- reg: means a register number between 0 and 31
- address/constant: means a 16-bit address/constant
- n.a.: means not applicable
- All the R-format instructions have the same value in the op-field. The hardware uses the funct-field to decide the variant of the R-type operation
- R-type and I-type instructions have similar formats with the same length

Translating MIPS Assembly Language into Machine Language

- `A[300] = h + A[300];`
  - h in $s2, base address of A in $t1
- Compiled MIPS code:

```
lw  $t0, 1200($t1)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

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

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
    - `sll $t2, $s0, 4`   # $t2 = $s0 << 4 bits
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operation

- 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 Operation

- 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
- In keeping with the 3-operand format, MIPS uses the NOR instruction instead of the NOT instruction
  - a NOR b == NOT (a OR b)
- `nor $t0, $t1, $t3        # $t0 = ~ ($t1 | $t3)`
- `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 |

Program Flow Control

- Decision making instructions
  - alter the control flow, i.e., change the "next" instruction to be executed
- Branch classifications
  - Unconditional branch
    - Always jump to the desired (specified) address
  - Conditional branch
    - Only jump to the desired (specified) address if the condition is true; otherwise, continue to execute the next instruction
- **Destination addresses** can be specified in the same way as other operands (*combination of register, immediate constant, and memory location*), depending on what addressing modes are supported in the ISA

MIPS Branch Operations

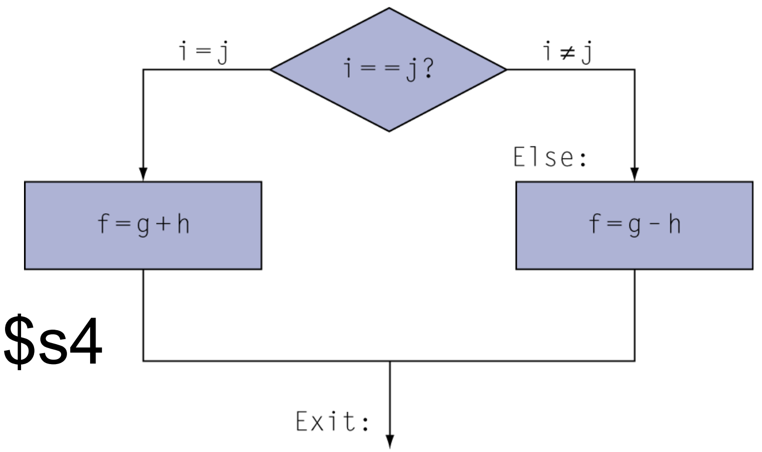
- Conditional branches
  - `beq rs, rt, L1`
    - if (rs == rt) branch to instruction labeled L1;
  - `bne rs, rt, L1`
    - if (rs != rt) branch to instruction labeled L1;
- Unconditional branches
  - `j L1`
    - unconditional jump to instruction labeled L1
  - `jal L1`
    - Jump and link
  - `jr $ra`
    - Jump register

Compiling If-then-else Statement

- C code:

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

  - f, g, h, i, j… in $s0, $s1, …, $s4



- Compiled MIPS code:

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

Assembler calculates addresses

Compiling a While Loop Statement

- C code:

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

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

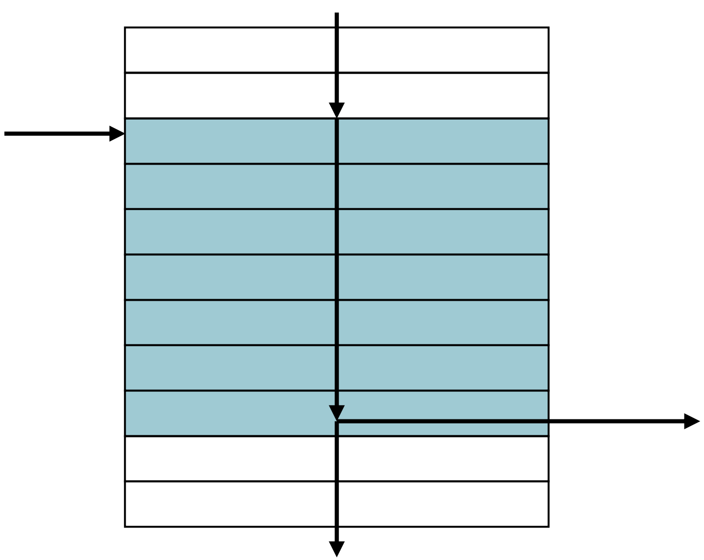
- Compiled MIPS code:

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

Why ?

The Basic Block

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



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

More Conditional Operations

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

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

- MIPS compiler uses the **`slt, beq, bne, $zero`** to create =, ≠, <, ≤, >, ≥

Branch Instruction Design

- **`beq`** and **`bne`** are the common case
- Why not **`blt`**, **`bge`**, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
  - MIPS compiler uses the **`slt, beq, bne, $zero`** to create =, ≠, <, ≤, >, ≥ is a good design compromise

Branches on LT/LE/GT/GE

- How to implement an equivalent `blt $s0, $s1, L1`?

```
slt $t0, $s0, $s1
bne $t0, $zero, L1    # $zero is always 0
```

- `bge  $s0, $s1, L1`?

```
slt  $t0, $s0, $s1
beq  $t0, $zero, L1
```

- `bgt  $s0, $s1, L1`?

```
slt  $t0, $s1, $s0
bne  $t0, $zero, L1
```

Try `ble` yourself !!

Signed vs. Unsigned Comparison

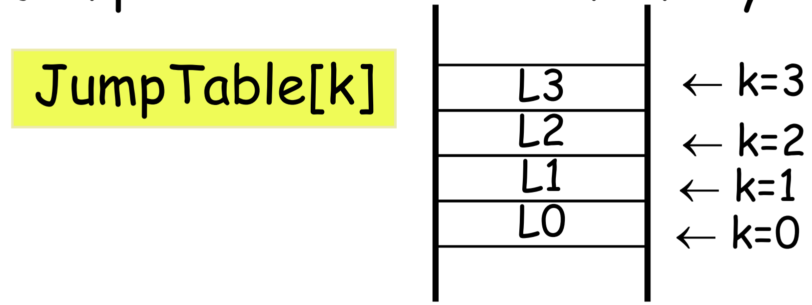
- Signed comparison: **`slt, slti`**
- Unsigned comparison: **`sltu, sltui`**
- Example
  - \$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - \$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - **`slt  $t0, $s0, $s1  # signed`**
    - $-1 < +1 \Rightarrow$ \$t0 = 1
  - **`sltu $t0, $s0, $s1  # unsigned`**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

Case/Switch Statement

- Case statement in C

```
switch (k){
  case 0: f=i+j;
  case 1: f=g+h;
  case 2: f=g-h;
  case 3: f=i-j;
}
```

Jump address table in memory



- A simplest way to implement case/switch is via a sequence of conditional tests, turning the case/switch statement into a chain of if-then-else statement
- One more efficient way is via a jump address table or jump table. And, the program needs only to index into the table and then jump to the appropriate label of sequence

Jump Register, jr

A switch statement for $0 \le k < 4$

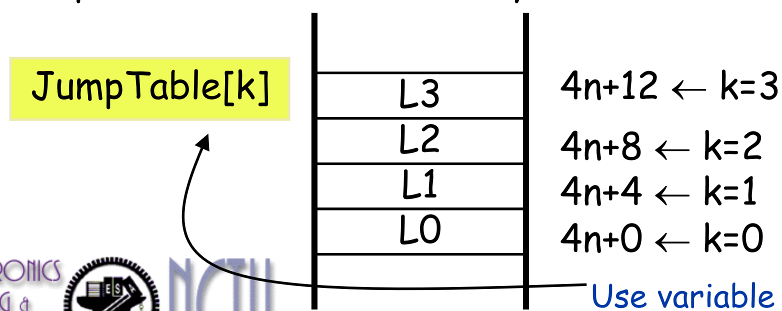
- Case statement in C

```
switch (k){
    case 0: f=i+j;
    case 1: f=g+h;
    case 2: f=g-h;
    case 3: f=i-j;
}
```

- Assume `f, g, h,i, j, k` are stored in registers `$s0`, `$s1`,…, and `$s5`, respectively
- Assume `$t2` contains 4
- Assume starting address contained in `$t4`, corresponding to labels L0, L1, L2, and L3, respectively

```
      slt  $t3, $s5, $zero   #test if k<0
      bne  $t3, $zero, Exit  #if k<0,exit
      slt  $t3, $s5, $t2     #test if k<4
      beq  $t3, $zero, Exit  #if k≥4,exit
-------------------------------------------
      add  $t1, $s5, $s5     #2k
      add  $t1, $t1, $t1     #$t1=4k
      add  $t1, $t1, $t4
      lw   $t0, 0($t1)
      jr   $t0
L0:add     $s0, $s3, $s4,
      j    Exit
L1:add     $s0, $s1, $s2
      j    Exit
L2:sub     $s0, $s1, $s2
      j    Exit
L3:sub     $s0, $s3, $s4
Exit:
```

Jump address table in memory



Use variable k to index a jump address tabke

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS
NCTU

Procedure Calling

- Steps required

Caller → Callee

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

Note that you have only one set of registers !!

Recall: Register Usage

- $a0 – $a3: arguments (reg's #4 – #7)
  - Used to pass parameters
- $v0, $v1: result values (reg's #2 and #3)
  - Used to return values
- $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)
  - Used to return to the point of origin

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  - Address of following instruction is saved 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

- MIPS code:

```
leaf_example:
   addi $sp, $sp, -4        Adjust stack for one item
   sw   $s0, 0($sp)         Save $s0 on stack
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3       Procedure body
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero     Result
   lw   $s0, 0($sp)         Restore $s0
   addi $sp, $sp, 4
   jr   $ra                 Return
```

Nested 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

A Recursive C 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

- 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    # if n≥1, go to 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
```

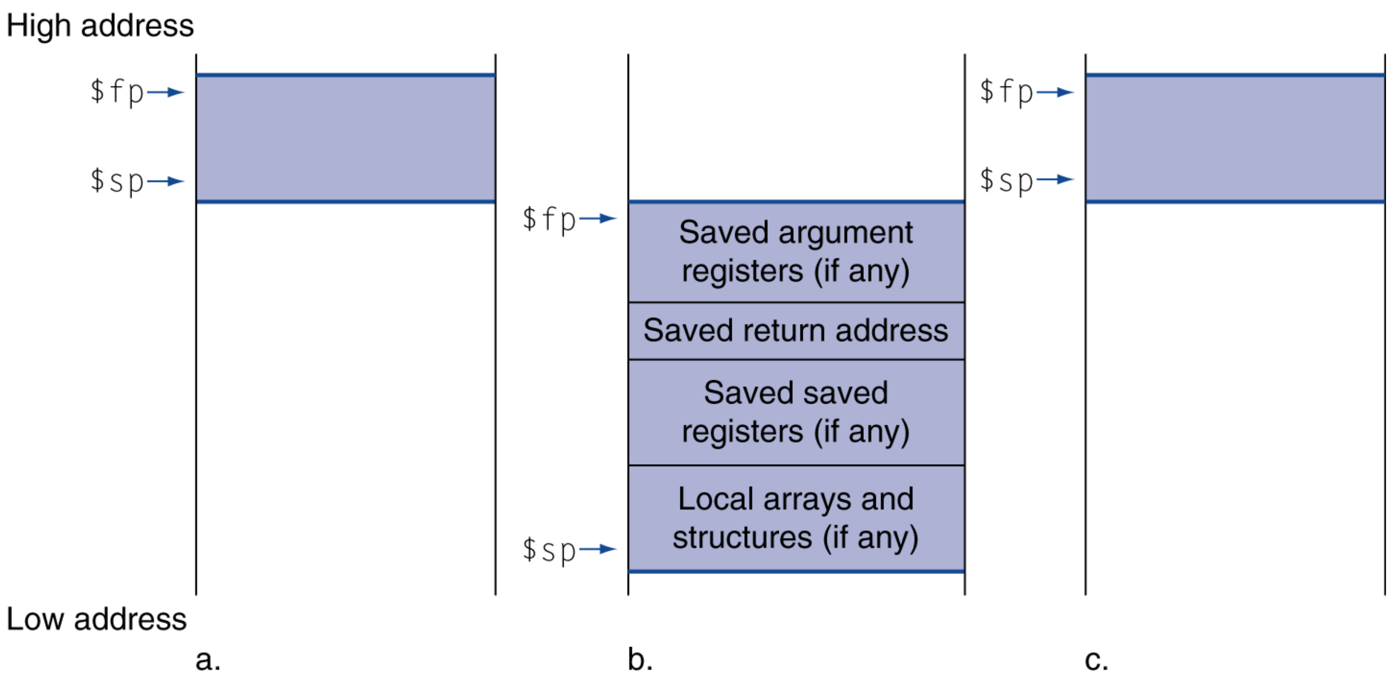
Remark

- What is and what is not preserved across a procedure call

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

  - $sp is itself preserved by the callee adding exactly the same amount that was subtracted from it
  - The other registers are preserved by saving them on the stack (if they are used) and restoring them from there

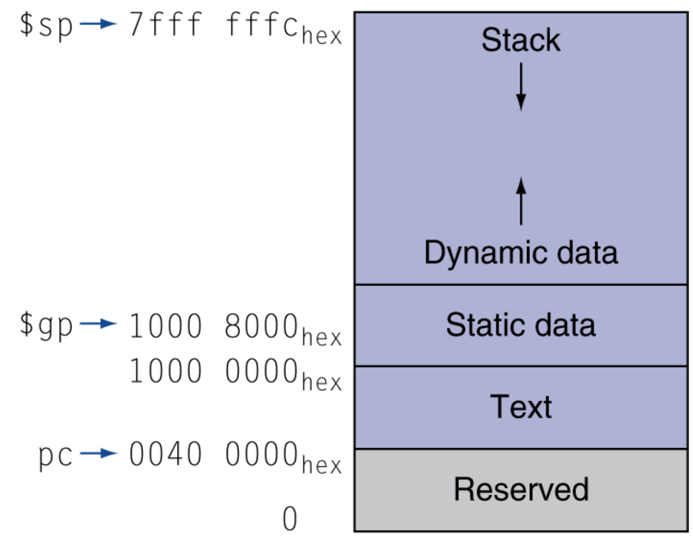
Local Data on the Stack



- Local data allocated by callee (local variables to the procedure, but do not fit in registers)
  - e.g., C automatic variables, arrays or structures, …
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: constants and other static (global) variables
  - e.g., static variables in C, constant arrays and strings
  - \$gp initialized to 1000 8000_H allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage
  - Start in the high end of memory and grows down
- Stack and heap are grown toward each other



Character Data

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

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case
  - Sign extend to 32 bits in rt

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

  - Zero extend to 32 bits in rt

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

  - Store just rightmost byte/halfword

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

String Copy Example

- C code (naïve):
  - Null-terminated string: used to mark the end of the 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 for i
     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

4000000 (22-bit)>16-bit

`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0011 1101	0000 1001 0000 0000

The Effect of the `lui` Instruction

The machine language version of `lui $t0, 255   # $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

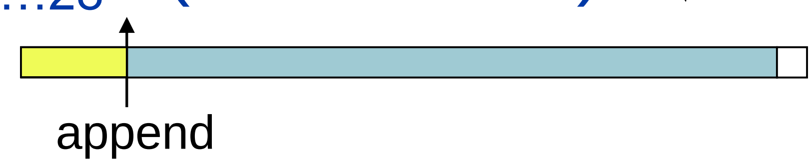
- Either the compiler or the assembler must break large constants into pieces and then resemble them into a register.
  - The immediate field's size is restricted
  - The assembler must have a temporary register available in which to create the long values for resembling them into a register.
  - That is why $at (assembler temporary) is reserved for the assembler.

Addressing in Jumps

`j    L1`

op	address
6 bits	26 bits

- Jump (**`j`** and **`jal`**) instruction is J-type
- The target address could be anywhere in text segment: Encode full address in instruction
- (Pseudo) Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address $\times$ 4)



Addressing in Conditional Branch

beq $t2, $zero, L2

op	rs	rt	constant or address (offset)
6 bits	5 bits	5 bits	16 bits

- Branch instructions specify: opcode, two registers, and target address
- Most target address is near to the PC
  - Forward or backward
- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Note: Word-alignment access

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						

Branching Far Away

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

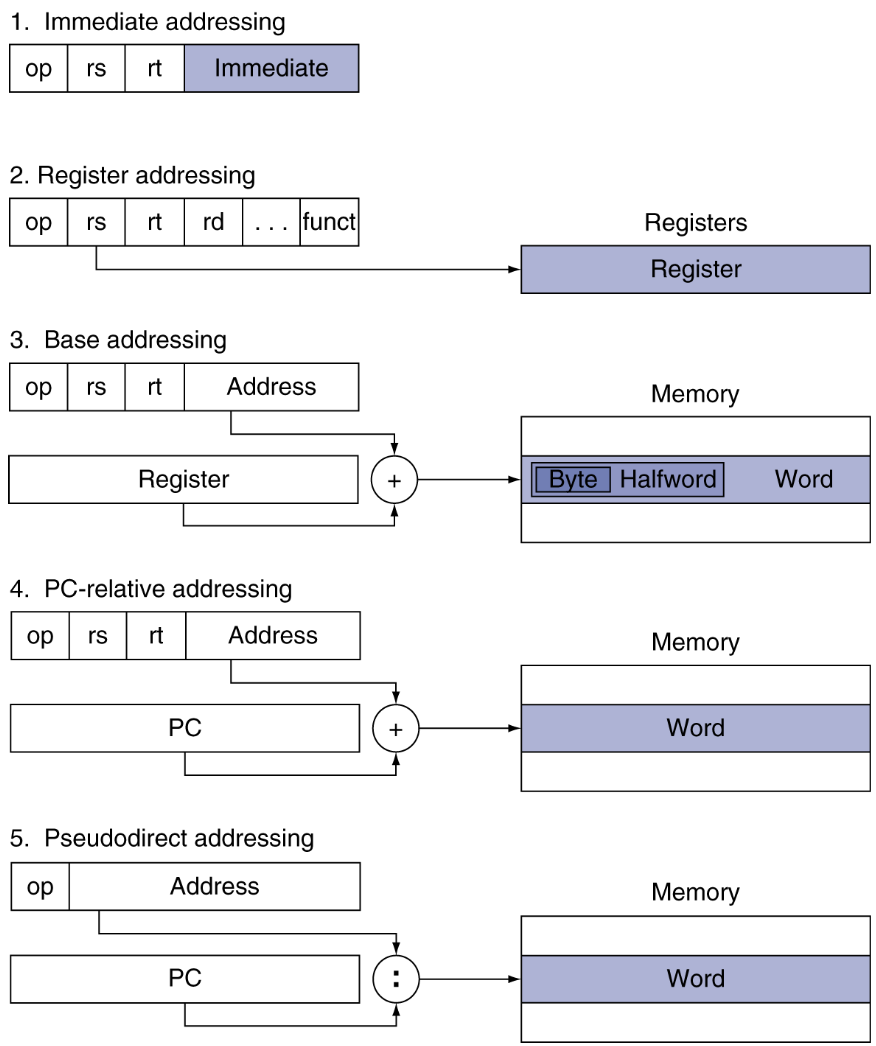
```
        beq $s0,$s1, L1
```

(larger than 16-bit offset)

↓

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

5 MIPS Addressing Modes



Decoding Machine Code

- Decoding: Reverse-engineer machine language to create the assembly language
- Example: 00af 8020hex

1. Convert hexadecimal to binary

   0000 0000 1010 1111 1000 0000 0010 0000

2. Look at the op field to determine the operation

   The op-field is 000000. It is an R-type instruction

3. Decode the rest of the instruction by looking at the field values

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

4. Reveal the assembly instruction

`add  $s0, $a1, $t7`

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Synchronization Issue

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

Synchronization in MIPS

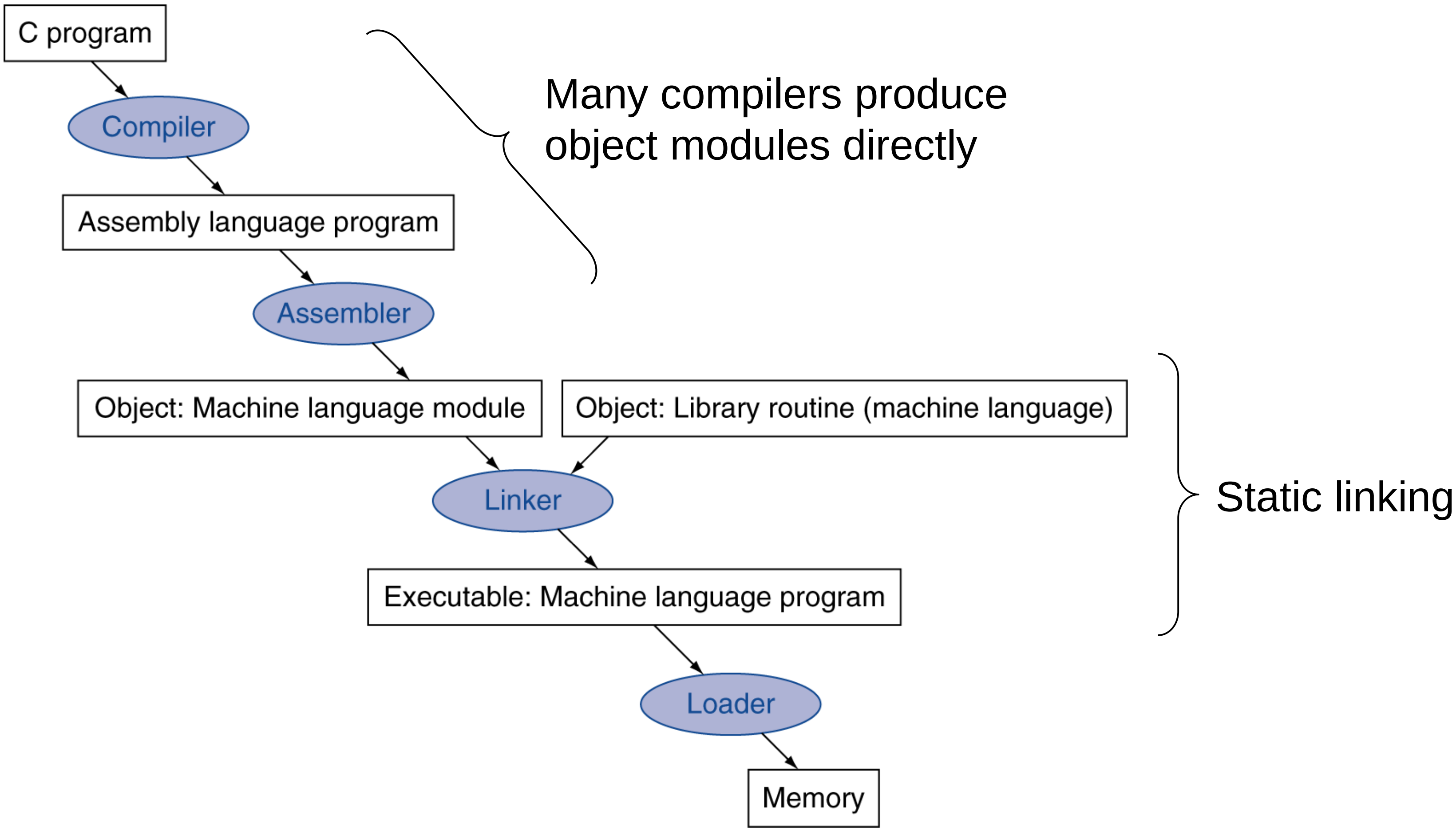
- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
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
```

lock-free atomic L/S

The contents of $s4 and the memory location specified by $s1 have been exchanged

Translation and Startup



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
```

- The cost of pseudoinstructions is reserving one register, **$at** (register 1): assembler temporary

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions and keeps track of labels used in branches and data transfer instruction in a symbol table.
- Object module provides information for building a complete program from the **six** distinct pieces (the object file for UNIX)
  - Header: used to describe the contents of the object module
  - Text segment: translated machine codes
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location when the program is loaded into memory
  - Symbol table: global definitions and external refs (or remaining labels) that are not defined
  - Debug info: for associating with source code

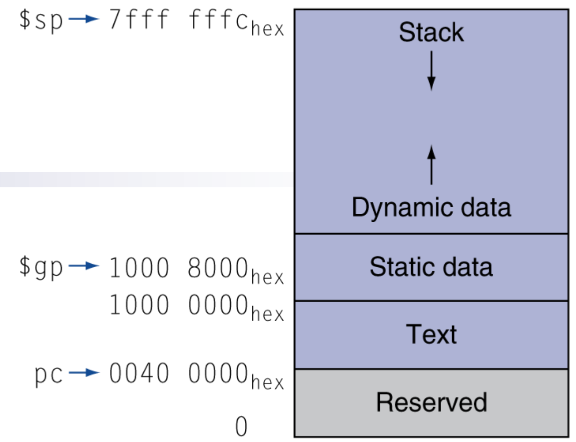
Linking Object Modules

- Linker: takes all the independently assembled program and stiches them together
- 3 steps for linker to produce an executable image
  1. Merges segments (i.e. place code and data modules symbolically in memory)
  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

Reading Assignment: P-133 Example

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	



Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	`lw $a0,` 8000_{hex}`($gp)`
	$0040\ 0004_{hex}$	`jal` $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	`sw $a1,` 8020_{hex}`($gp)`
	$0040\ 0104_{hex}$	`jal` $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create (virtual) address space, which is large enough for the text and data
  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, if necessary
  5. Initialize registers (including $sp, $fp, $gp to the first free location)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit system-call

Dynamic Linking

- Static linking problem
  - The library routines become part of the executable code. It keeps using the old version of the library even though a new one is released.
  - It loads all routines in the library that are called anywhere in he executable, even if those calls are not executed.
- Dynamically linked libraries (DLL): 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

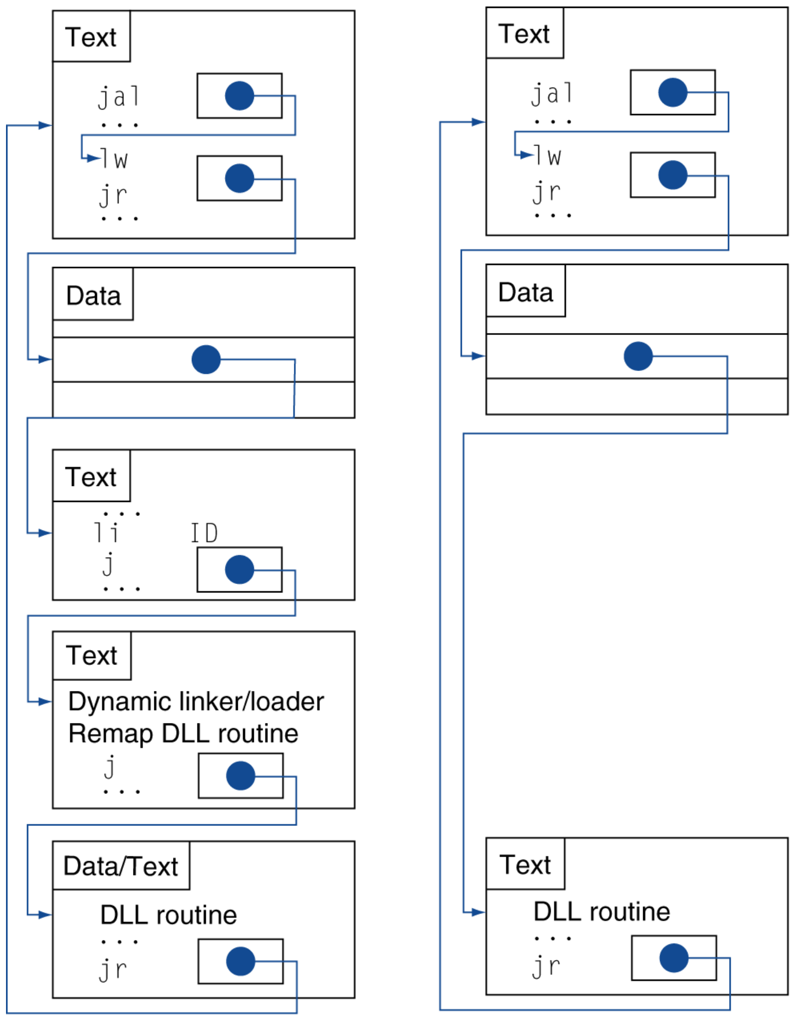
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

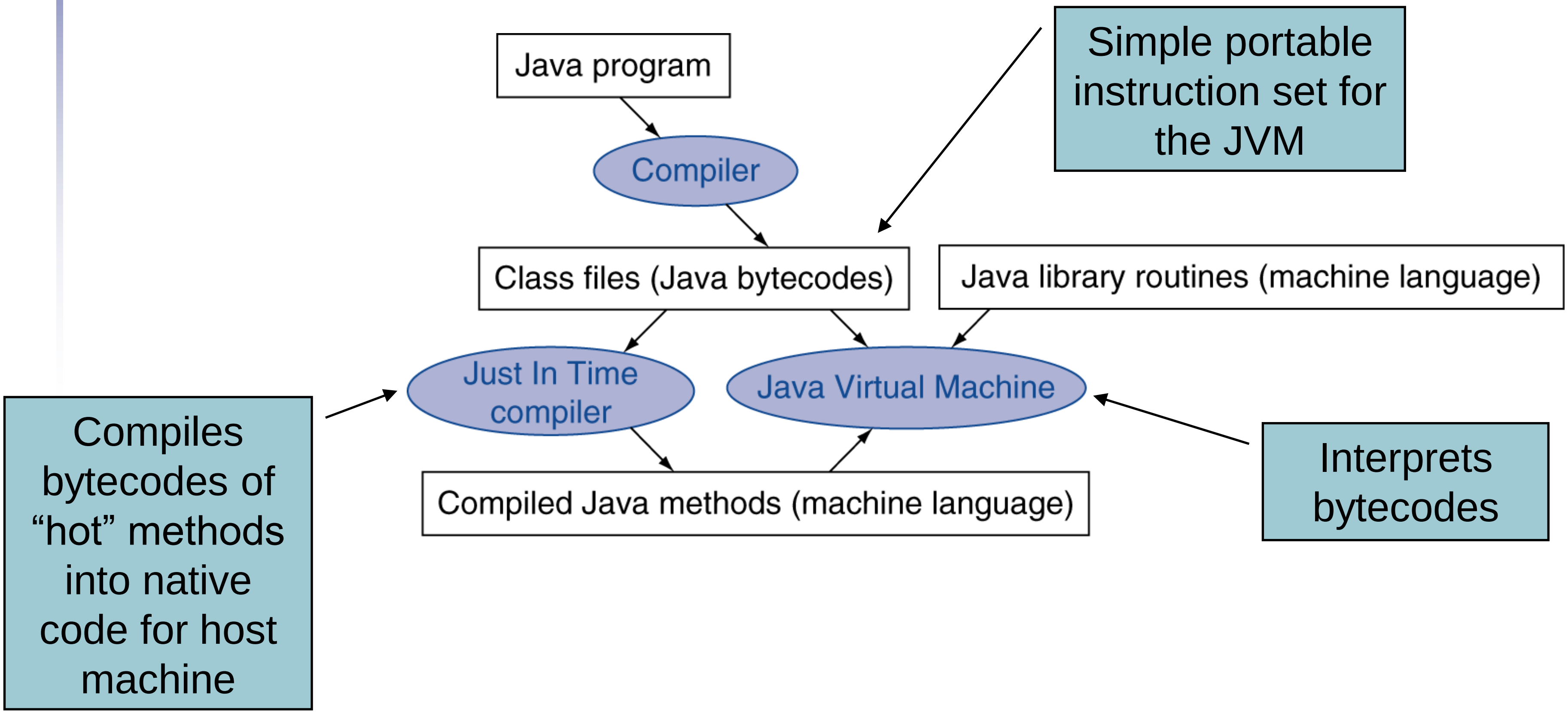
Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Starting a Java Program



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

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 $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         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
         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
         addi $s1, $s1, –1      # j –= 1
         j    for2tst           # jump to test of inner loop
exit2:   addi $s0, $s0, 1       # i += 1
         j    for1tst           # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

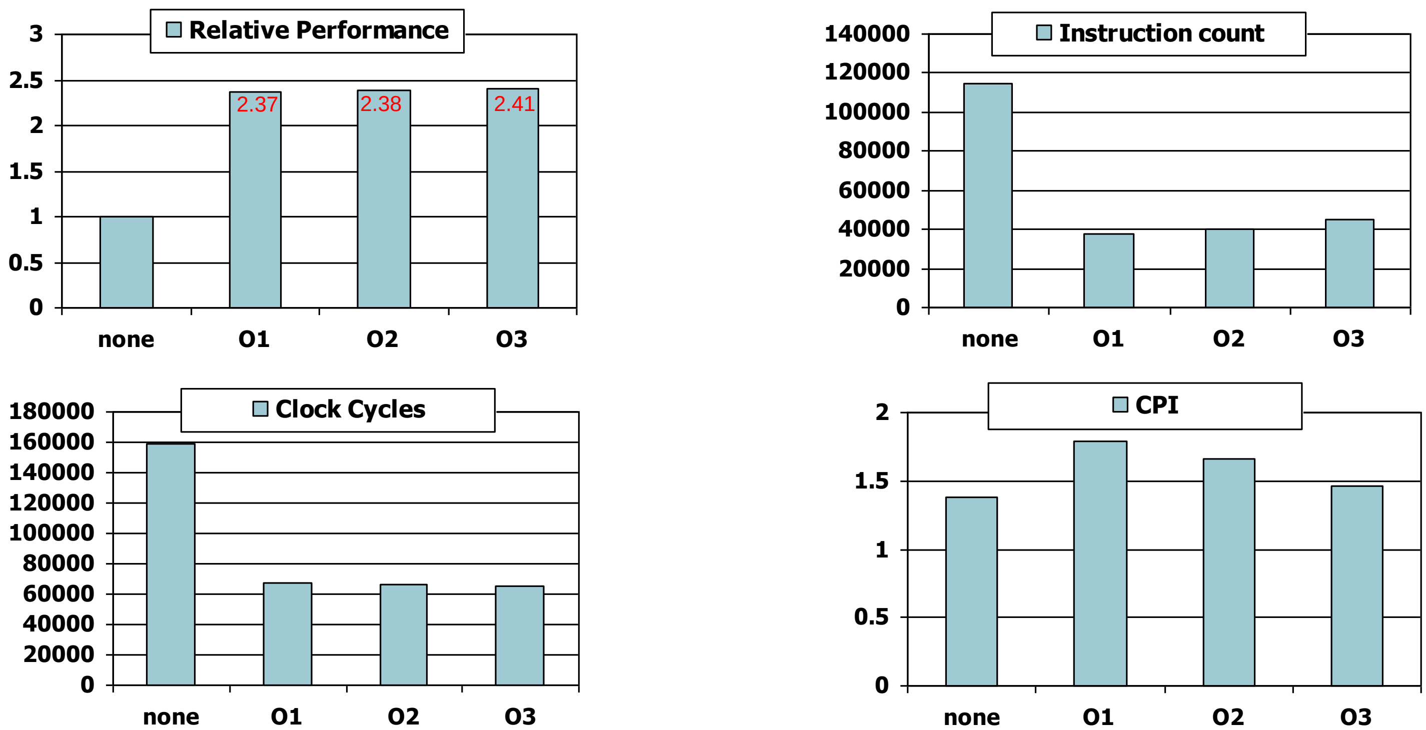
Outer loop

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
```

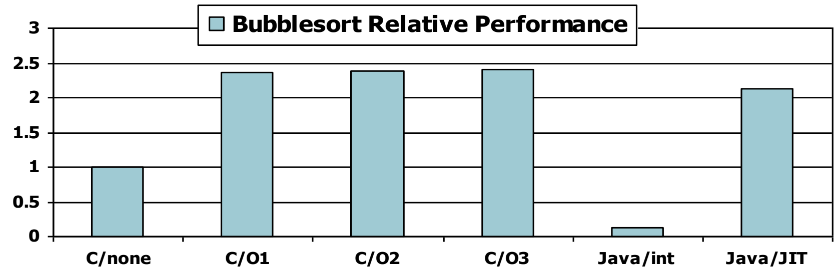
Effect of Compiler Optimization

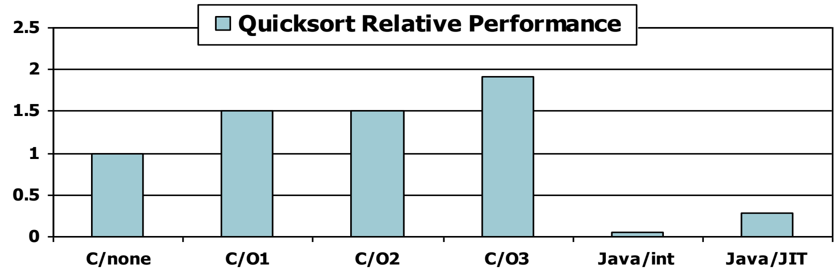
Compiled with gcc for Pentium 4 under Linux

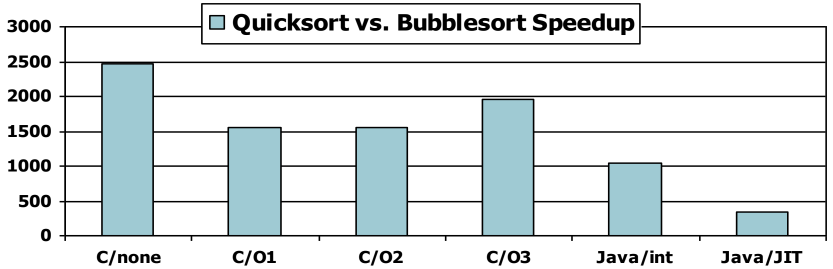


- Un-optimized code has the best CPI
- O1 optimization has the lowest instruction count
- O3 optimization is the fastest

Impact of Language and Algorithm







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!

Arrays vs. Pointers

- A challenge for new C programmer is understanding *pointers*.
- Two C examples: array indices vs. pointers

`clear1(int array[], int size) {` `  int i;` `  for (i = 0; i < size; i += 1)` `    array[i] = 0;` `}`	`clear2(int *array, int size) {` `  int *p;` `  for (p = &array[0]; p < &array[size];` `       p = p + 1)` `    *p = 0;` `}`

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

Example of Clearing with Array vs. Pointer

<pre>clear1(int array[], int size) { int i; for (i = 0; i < size; i += 1) array[i] = 0; }</pre>	<pre>clear2(int *array, int size) { int *p; for (p = &array[0]; p < &array[size]; p = p + 1) *p = 0; }</pre> Assign pointer **p** to the address of the first element
<pre> 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</pre>	<pre> move $t0,$a0 # p = & array[0] loop2: sw $zero,0($t0) # Memory[p] = 0 addi $t0,$t0,4 # p = p + 4 sll $t1, $a1, 2 # $t1 = size * 4 add $t2, $a0, $t1 # $t2 = # address of array[size] slt $t3,$t0,$t2 # $t3 = #(p<&array[size]) bne $t3,$zero,loop2 # if (…) # goto loop2</pre>

- We assume that the two parameters **`array`** and **`size`** are found in the registers **`$a0`** and **`$a1`**

Fast Version of `clear2`

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

Always the same

```
       move $t0,$a0
       sll $t1,$a1,2
       add $t2,$a0,$t1
loop2: sw $zero,0($t0)
       addi $t0,$t0,4
       slt $t3,$t0,$t2
       bne $t3,$zero,loop2
```

Comparing the Two Versions of Clear

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

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

```
       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
```

- Array indices method must calculate the address of the new index "**`i`**"
- Pointer method increments the pointer "**`p`**" directly

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

ARM & MIPS Similarities

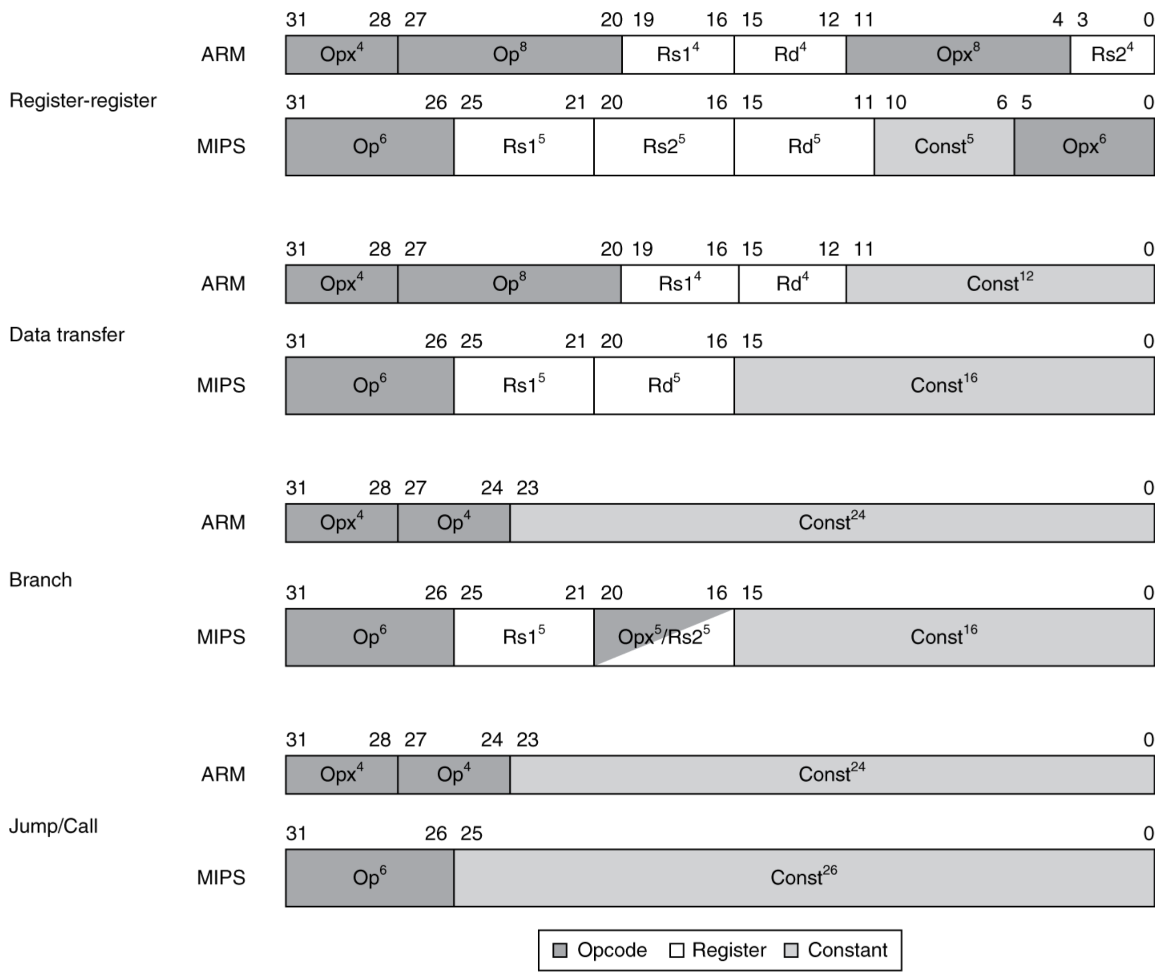
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	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

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

Instruction Encoding



ARM v8 Instructions

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

RISC-V Instructions

- Most similar to MIPS.
- An open architecture

Register-register

	31–25	24–20	19–15	14–12	11–7	6–0
RISC-V	funct7(7)	rs2(5)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31–26	25–21	20–16	15–11	10–6	5–0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Const(5)	Opx(6)

Load

	31–20	19–15	14–12	11–7	6–0
RISC-V	immediate(12)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31–26	25–21	20–16	15–0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Store

	31–25	24–20	19–15	14–12	11–7	6–0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31–26	25–21	20–16	15–0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Branch

	31–25	24–20	19–15	14–12	11–7	6–0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31–26	25–21	20–16	15–0
MIPS	Op(6)	Rs1(5)	Opx/Rs2(5)	Const(16)

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	ld x5, 40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

Common Features between RISC-V and MIPS

- All instructions are 32-bit wide for both architectures
- Both have 32 general-purpose registers
- The only way to access memory is via load and store instructions on both architectures
- There are no instructions that can load or store many registers in MIPS or RISC-V
- Both have instructions that branch if a register is equal to zero and branch if a register is not equal to zero
- Both sets of addressing modes work for all data sizes

The Intel x86 ISA

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

The Intel x86 ISA

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

The Intel x86 ISA

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

Basic x86 Registers

Name (31–0)	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

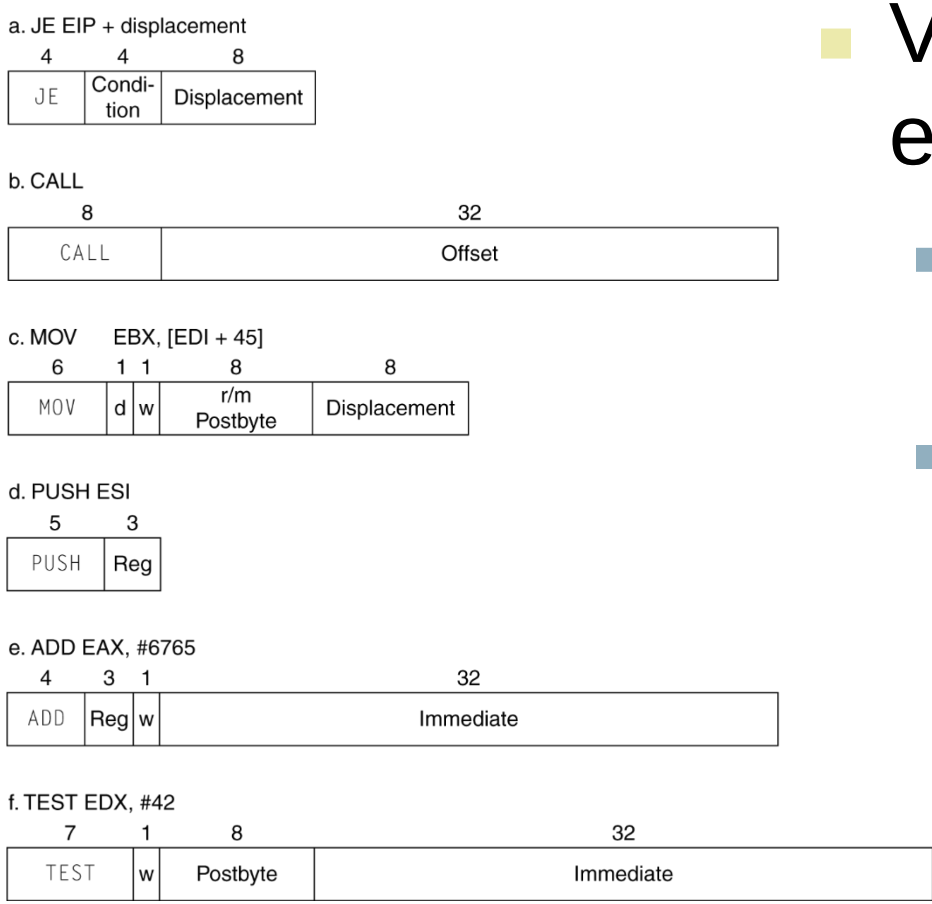
Basic x86 Addressing Modes

- 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

x86 Instruction Encoding



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

Implementing IA-32

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

ARM v8 Instructions

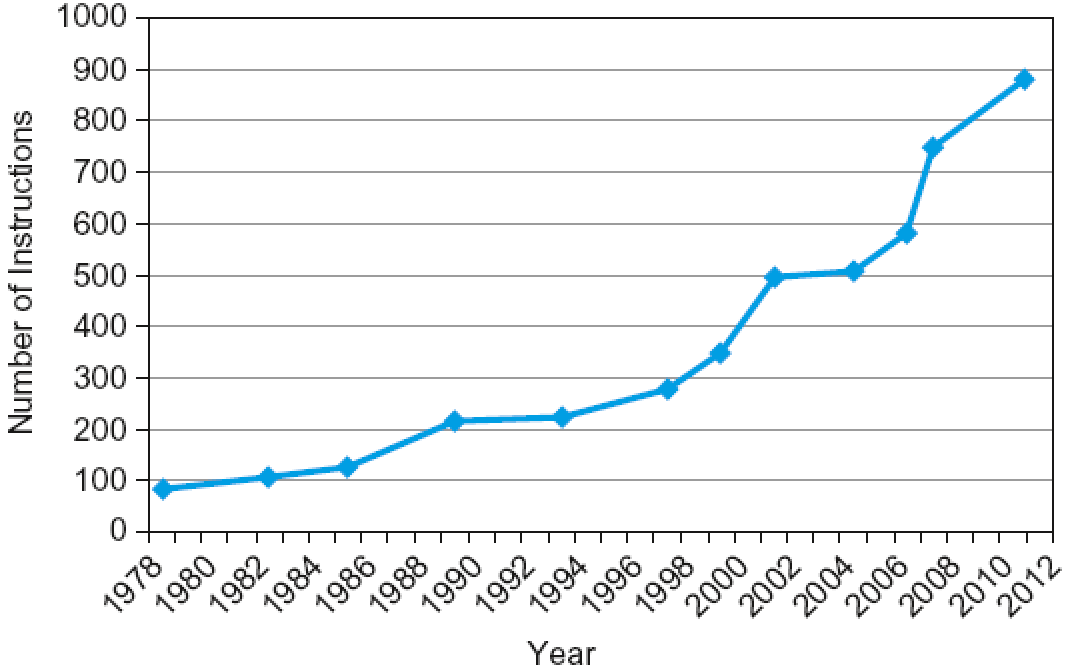
- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all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

Fallacies

- Powerful instruction $\Rightarrow$ 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 $\Rightarrow$ more errors and less productivity

Fallacies

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



x86 instruction set

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

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%