Computer Organization and Design (Spring 2021)

Instructions: Language of the Computer

Jiang Zhong
zhongjiang@cqu.edu.cn
qq: 376917902

Instruction Set

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

The MIPS Instruction Set

- Used as the example throughout the course
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in smartphones, tablets, consumer electronics, network/storage equipment, cameras, printers, …

投票(匿名) 最多可选3项

设置

同学们之前阅读机器代码的经历

A C/C++语义的集成开发环境中

B 可执行程序的反汇编工具

C 其它二进制编辑器

D 没有看过机器代码

提交

投票(匿名) 最多可选4项

⚙ 设置

关于机器指令的知识

- A 了解80x86机器指令类型和编码格式
- B 知道计算机内部的寻址方式
- C 熟悉机器指令编码方案
- D 了解RISC-V开源指令集的特点

提交

Arithmetic Operations

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

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

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

§ 2.2 Operations of the Computer Hardware

Arithmetic Example

- 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
```

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
  - Reason for small register file
  - 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
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data, …
  - Only a small amount of data can fit in registers
- 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

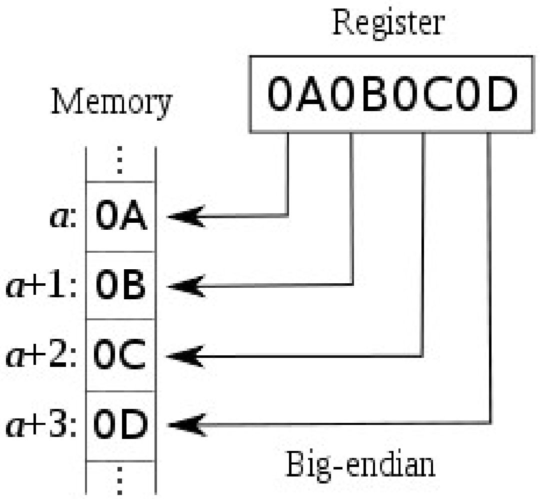
Big vs. Little Endian

- Big Endian: leftmost byte is word address

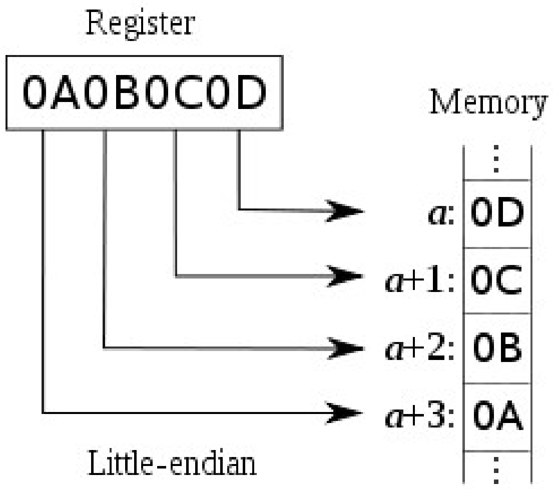
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

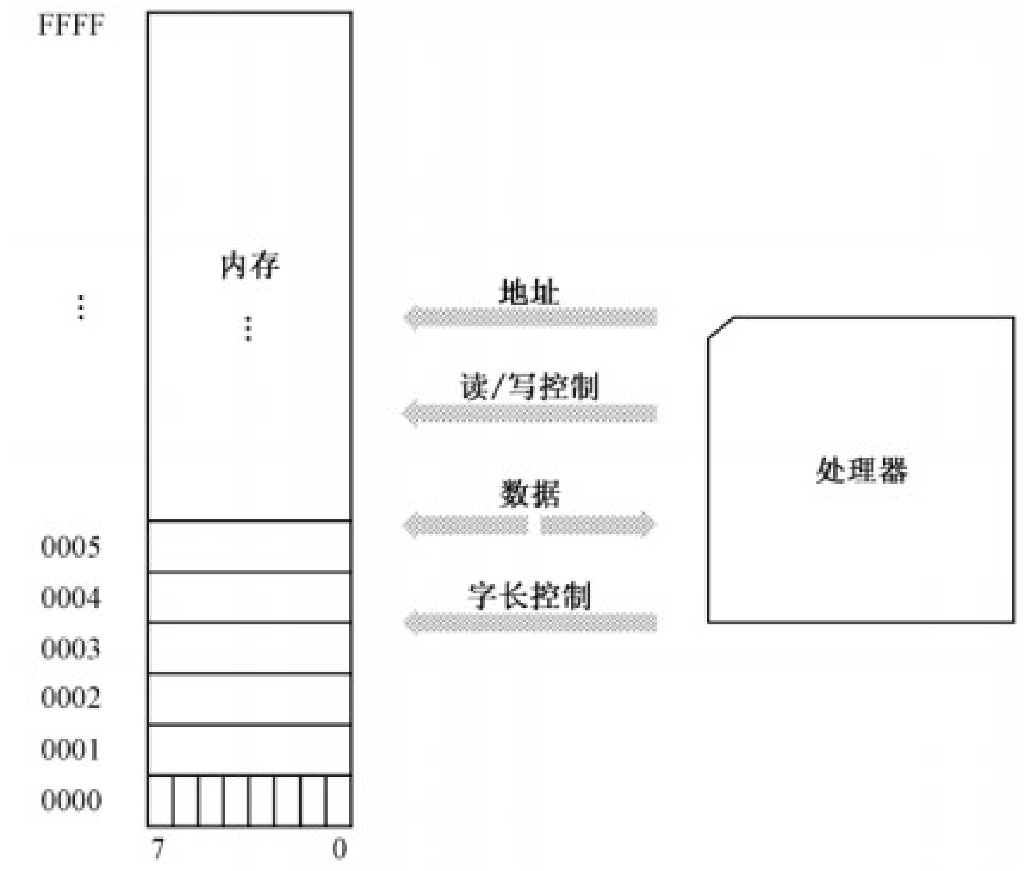


Big-endian



Little-endian

Memory vs. CPU



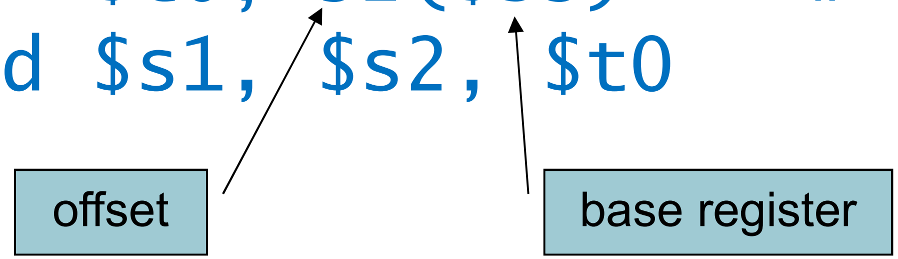
Memory Operand Example 1

- C code:

  `g = h + A[8];`

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

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



offset

base register

Memory Operand Example 2

- C code:

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

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

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

Registers vs. Memory

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

Immediate Operands

- Constant data specified in an instruction

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

- No subtract immediate instruction
  - Just use a negative constant

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

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Number types

- Integer numbers, unsigned
  - Address calculations
  - Numbers that can only be positive
- Signed numbers
  - Positive
  - Negative
- Floating point numbers
  - numeric calculations
  - Different grades of precision
    - Singe precision (IEEE 754)
    - Double precision (IEEE 754)
    - Quadruple precision

Number formats

- Sign and magnitude
- 2's complement
- 1's complement

  similar to 2's complement, + 0 & - 0
- Biased notation

  1000 0000 = minimal negative value(-2^7)

  0111 1111 = maximal positive value (2^7-1)
- Representation
  - Binary
  - Decimal
  - Hexadecimal

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

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
addi $t2, $zero,100
```

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

§2.4 Signed and Unsigned Numbers

Signed number representation

- First idea:

  Positive and negative numbers

  - Take one bit (e.g. 31) as the **sign bit**
    - Problem
    - **0** 0000000 = 0 positive zero!
    - **1** 0000000 = 0 negative zero!
  - Each comparison to 0 requires two steps
- 1's complement
- 2's complement

2s-Complement Signed Integers

- Given an n-bit number

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

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

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111
- By default, all data in MIPS is a signed integer
  - add vs. addu

Signed Negation

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

$$x + \overline{x} = 1111\ldots111_2 = -1$$

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

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

Sign Extension

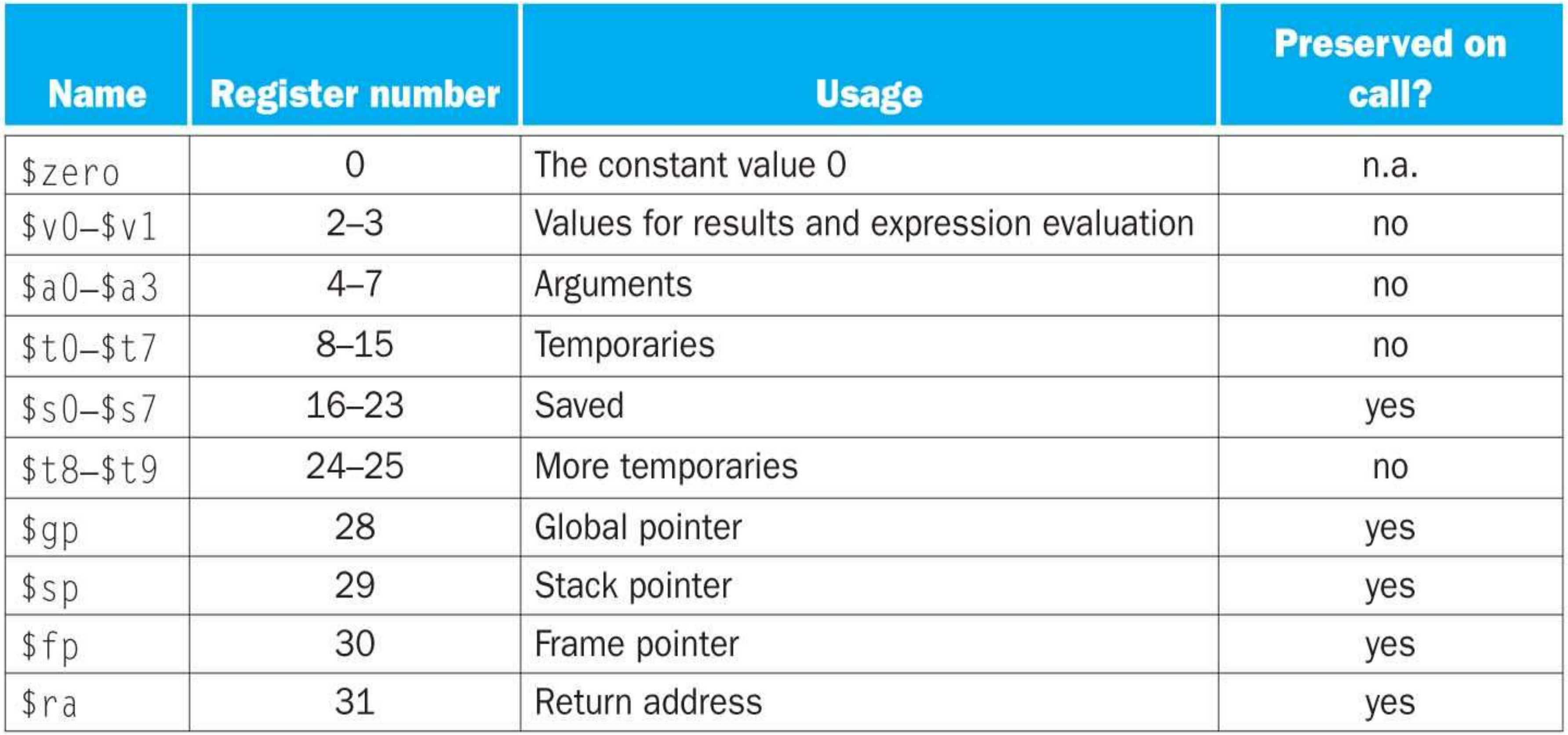
- How to represent a number using more bits?
  - e.g. 16 bit number as a 32 bit number
  - Must preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement

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

§ 2.5 Representing Instructions in the Computer

MIPS Register Conventions



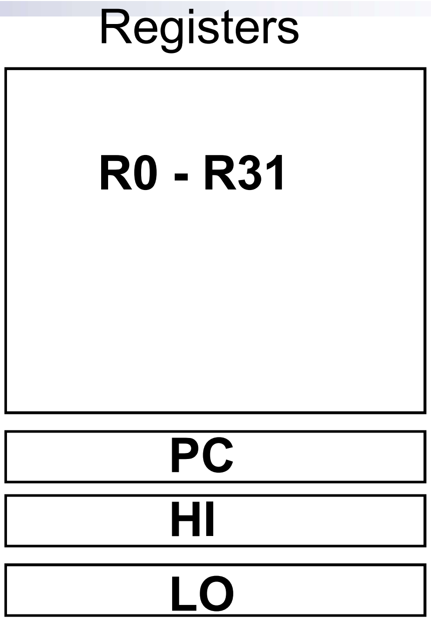
Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes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

Register 1, called $at, is reserved for the assembler and registers 26−27, called $k0−$k1, are reserved for the operating system.

MIPS-32 ISA

- Instruction Categories
  1. Computational
  2. Load/Store
  3. Jump and Branch
  4. Floating Point

     --coprocessor
  5. Memory Management
  6. Special



Registers

R0 - R31

PC

HI

LO

Instruction Formats: **all 32 bits wide**

op	**rs**	**rt**	**rd**	**sa**	**funct**	R format
op	**rs**	**rt**	**immediate**			I format
op	**jump target**					J format

MIPS R-format Instructions

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

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

R-format Example

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

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

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

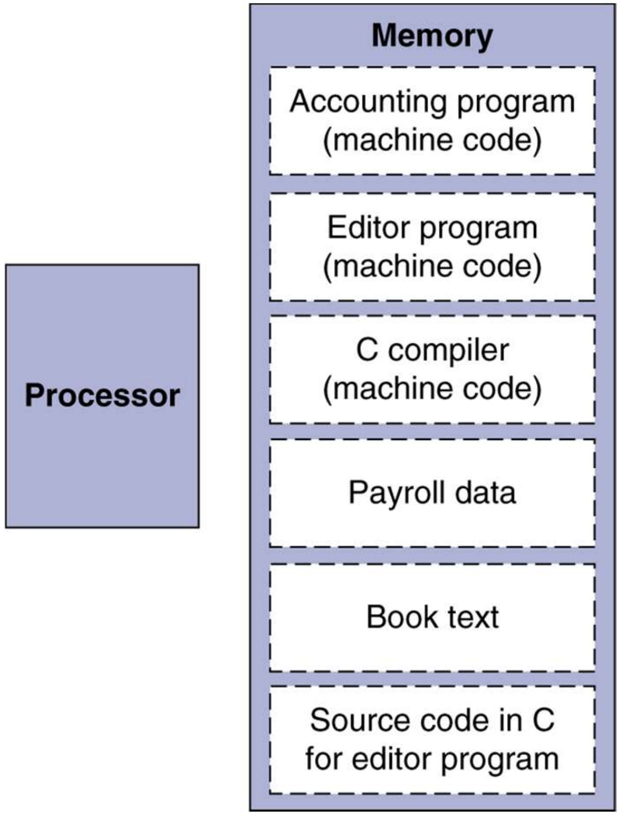
Two Key Principles of Computer Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data

2. Programs are stored in alterable memory (that can be read or written to) just like data

The BIG Picture

- Stored-program concept
  1. Programs can be shipped as files of binary numbers – binary compatibility
  2. Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs


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

单选题 1分

设置

请问计算机内的数据和指令表示正确是

[✓ A] 指令和数据都是采用二进制

B 指令采用十进制，数据采用二进制

C 指令和数据都采用十进制

提交

单选题 1分

设置

两个计算机系统的指令集架构完全相同，如果可执行程序在其中一个系统能够正确运行，那么一定能够在另外一台计算机上运行。

A 正确

[✓ B 错误]

提交

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

Shift Operations

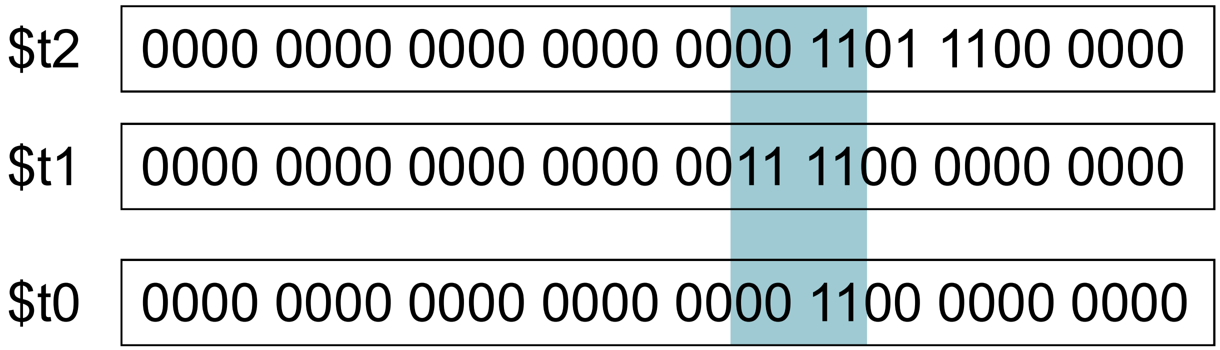
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```



$t2 0000 0000 0000 0000 0000 1101 1100 0000

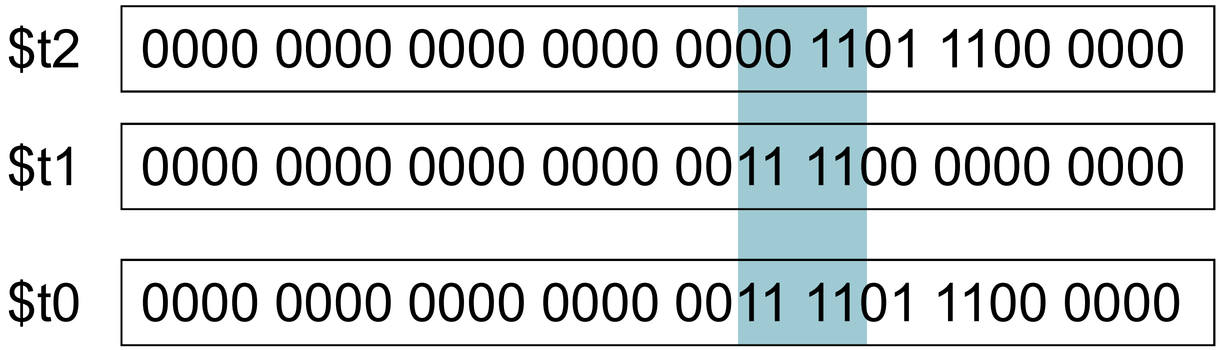
$t1 0000 0000 0000 0000 0011 1100 0000 0000

$t0 0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```



$t2 0000 0000 0000 0000 0000 1101 1100 0000

$t1 0000 0000 0000 0000 0011 1100 0000 0000

$t0 0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR 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

单选题 1分

设置

请问可将寄存器的特定的几位设置为1的MIPS指令为

A AND

[✓ B] OR

C NOT

D SLL

提交

单选题 1分

设置

计算机中的逻辑运算指令除了AND,OR,NOT指令之外是否一定需要使用XOR,NOR等其他逻辑运算指令才能满足任意逻辑表达式的需要

A 是，需要实现各种逻辑运算对应的指令

[✓ B] 否，只需要AND,OR,NOT三条指令即可实现任意的逻辑表达式

C AND,OR,NOT不一定够，还需要其它的逻辑运算指令

提交

Conditional Operations

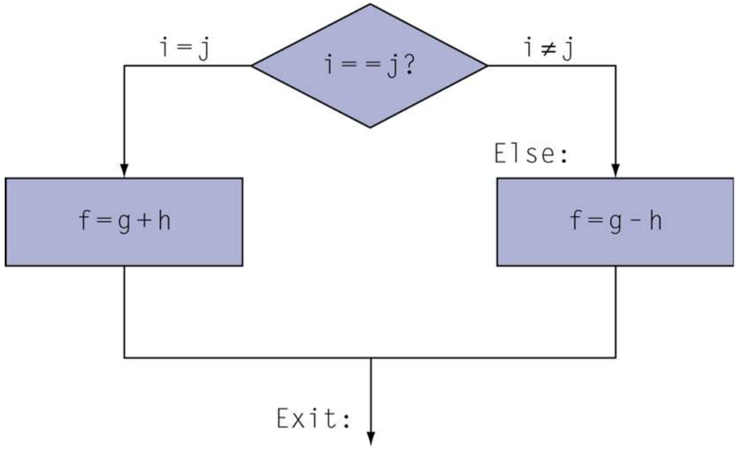
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

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  # $t1 = i*4
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

More Conditional Operations

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

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

Branch Instruction Design

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

主观题 2分

设置

请基于MIPS的分支指令实现以下C语言程序

```
 if （i>=j）
   i=i+1;
 else
   j=j+1;
```

假定变量i,j分别使用寄存器$s1和$s2，其它的寄存器建议使用任意临时寄存器。

正常使用主观题需2.0以上版本雨课堂

作答

Signed vs. Unsigned

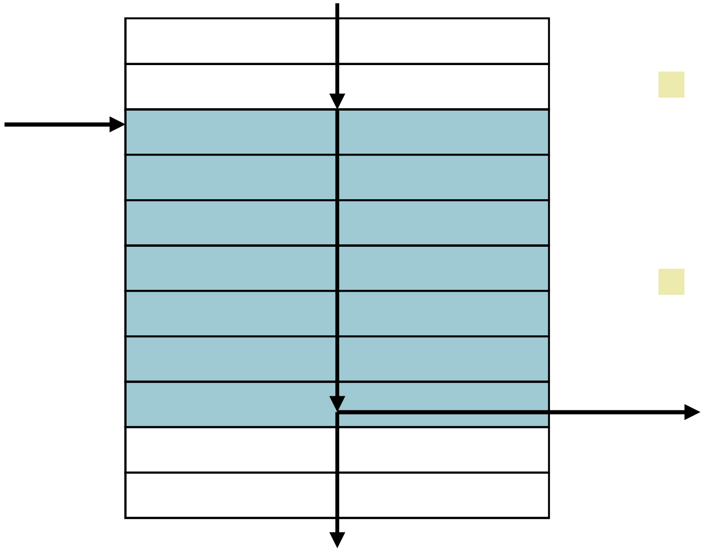
- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295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

Register Usage

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

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

Procedure Calling

- Procedures enable structured programs
  - Easier to understand and reuse code
- 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

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

主观题 1分

设置

课堂讨论：为什么在程序中使用自定义函数?

正常使用主观题需2.0以上版本雨课堂

作答

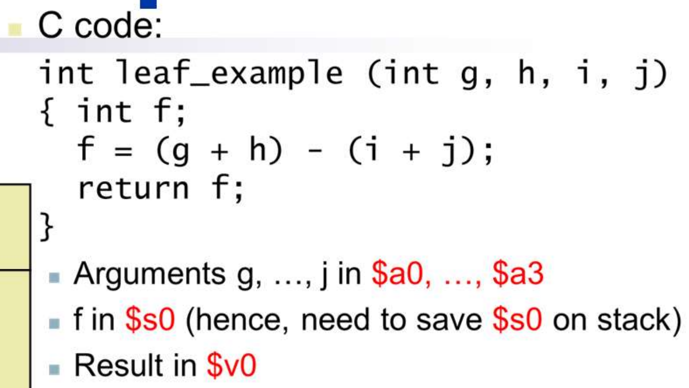
Leaf Procedure Example

- 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;
}
```

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

- MIPS code:

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

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Think about the differences between 80x86 and MIPS Procedure Instructions!

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Any saved registers being used $s0-$s7
  - Its return address
  - Any arguments and temporaries needed after the call
    - e.g., suppose main program calls procedure A with an argument 3 in $a0 and then using jal A
    - Then if procedure A calls procedure B via jal B with an argument of 7 placed in $a0, there is a conflict over the use of register $a0
- 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);
}
```

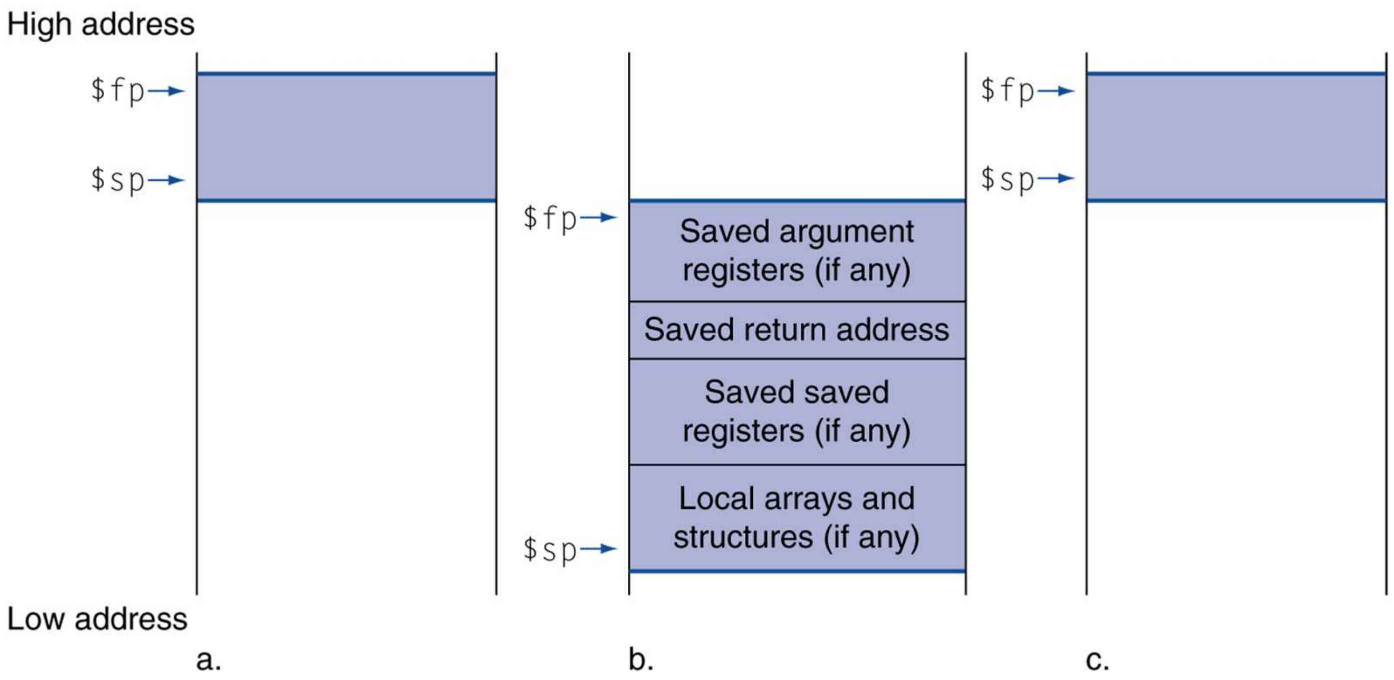
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- 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
```

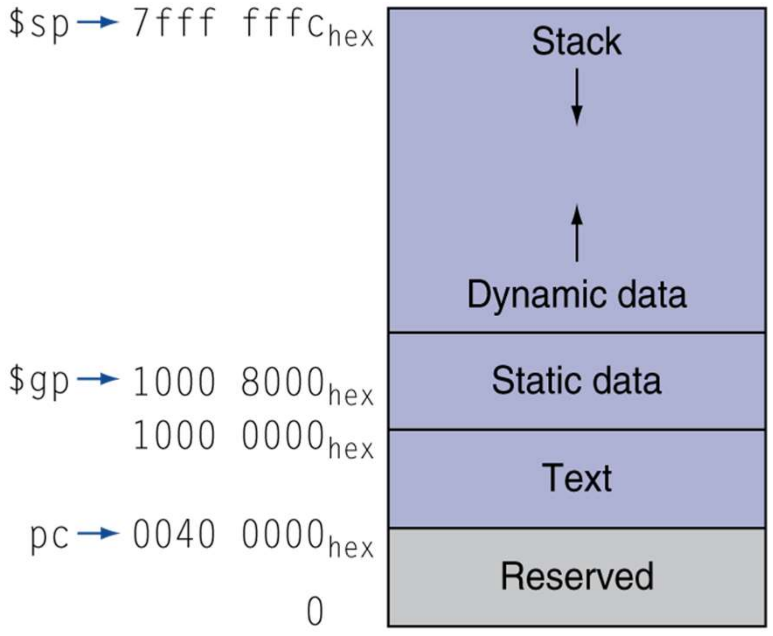
Local Data on the Stack


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 (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, linked lists
  - Must use free() in C
    - Memory leak issues!
- Stack: automatic storage



单选题 1分 ⚙ 设置

递归函数是否可以转换为循环结构的非递归程序?

[✓ A] 可以

B 部分可以

C 不行

提交

主观题 2分

设置

课堂研讨：

对比MIPS和Intel 80x86指令集中的函数调用和返回的指令，并分析各自的优缺点。

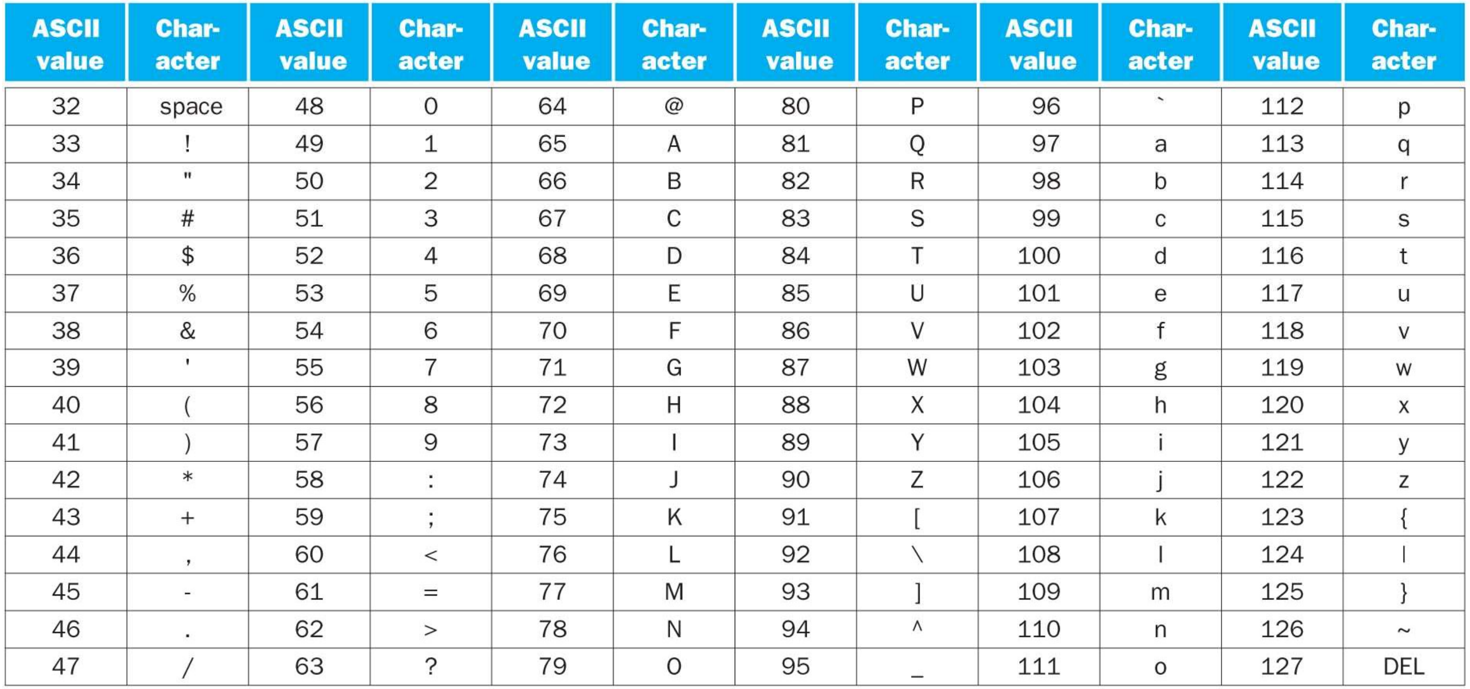
可以在网上查找并对比

作答

Character Data

§ 2.9 Communicating with People

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 non-printable control (almost obsolete now)

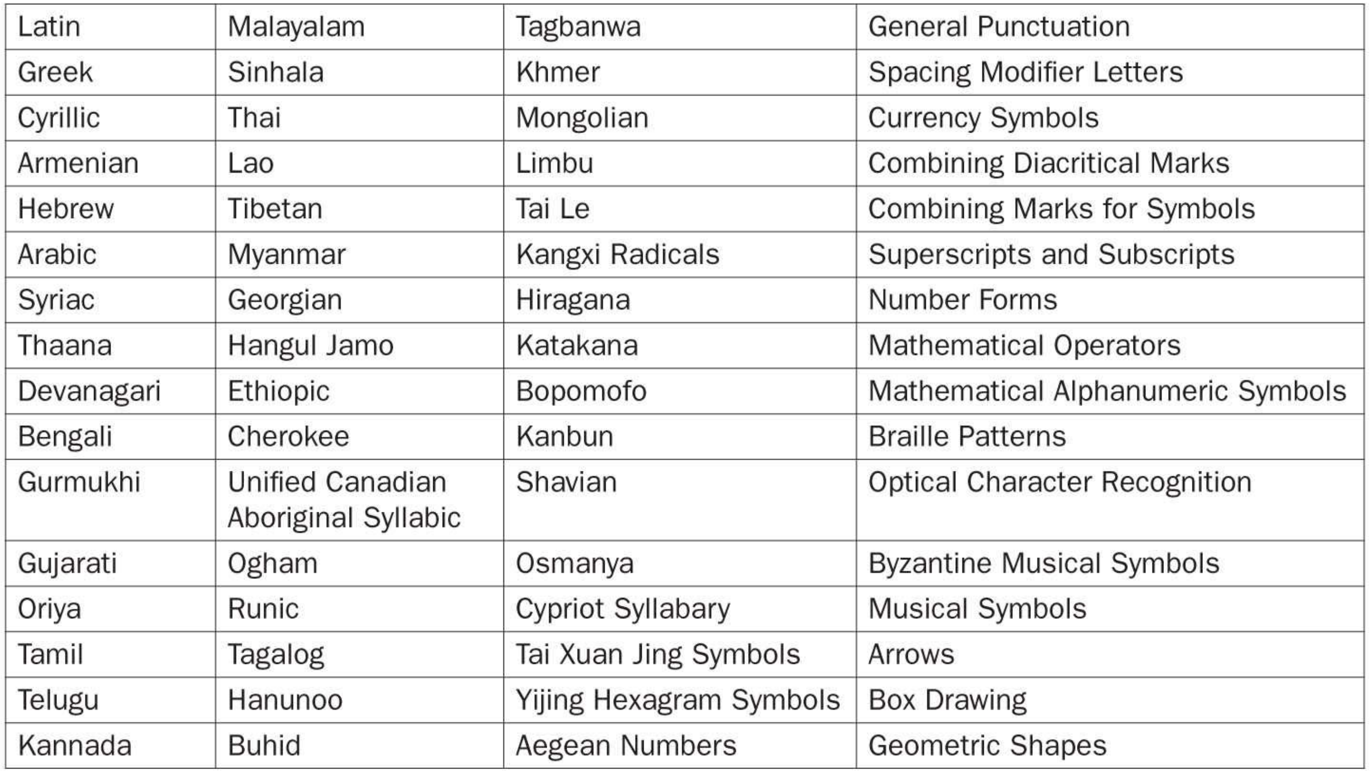


ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

- Latin-1: 256 characters
  - ASCII, +96 more graphic characters

Character Data

- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - UTF-8, UTF-16: variable-length encodings
  - Most of the world’s alphabets, plus symbols



Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

Byte/Halfword Operations

- To manipulate text (e.g. ASCII characters) byte operations are essential
- 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**

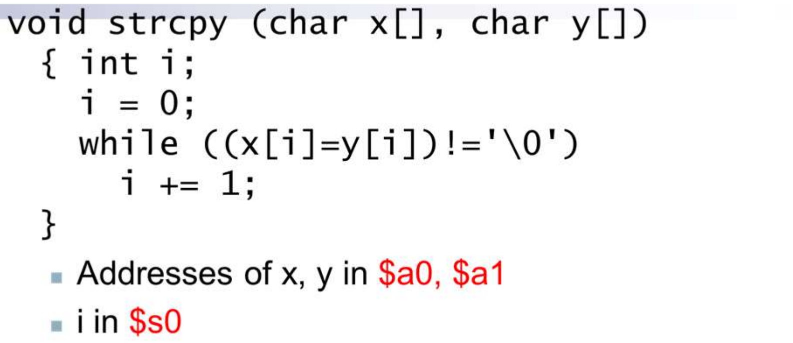
String Copy Example

- C code (naïve):
  - 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



```
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

- 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`

  - Copies 16-bit const to left 16 bits of rt; Clears right 16 bits of rt
- Example: how can we load 32 bit constant into $s0?

  0000 0000 0111 1101 0000 1001 0000 0000

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

- Typically assembler is responsible for breaking a large constant and reassembling into a register (using the reserved register $at)

§ 2.10 MIPS Addressing for 32-Bit Immediates and Addresses

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- If addresses had to fit in 16-bits, then no program could be bigger than 2^{16}

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
  - Almost all loops and *if* statements < 2^{16} words

Jump Addressing

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

op	address
6 bits	26 bits

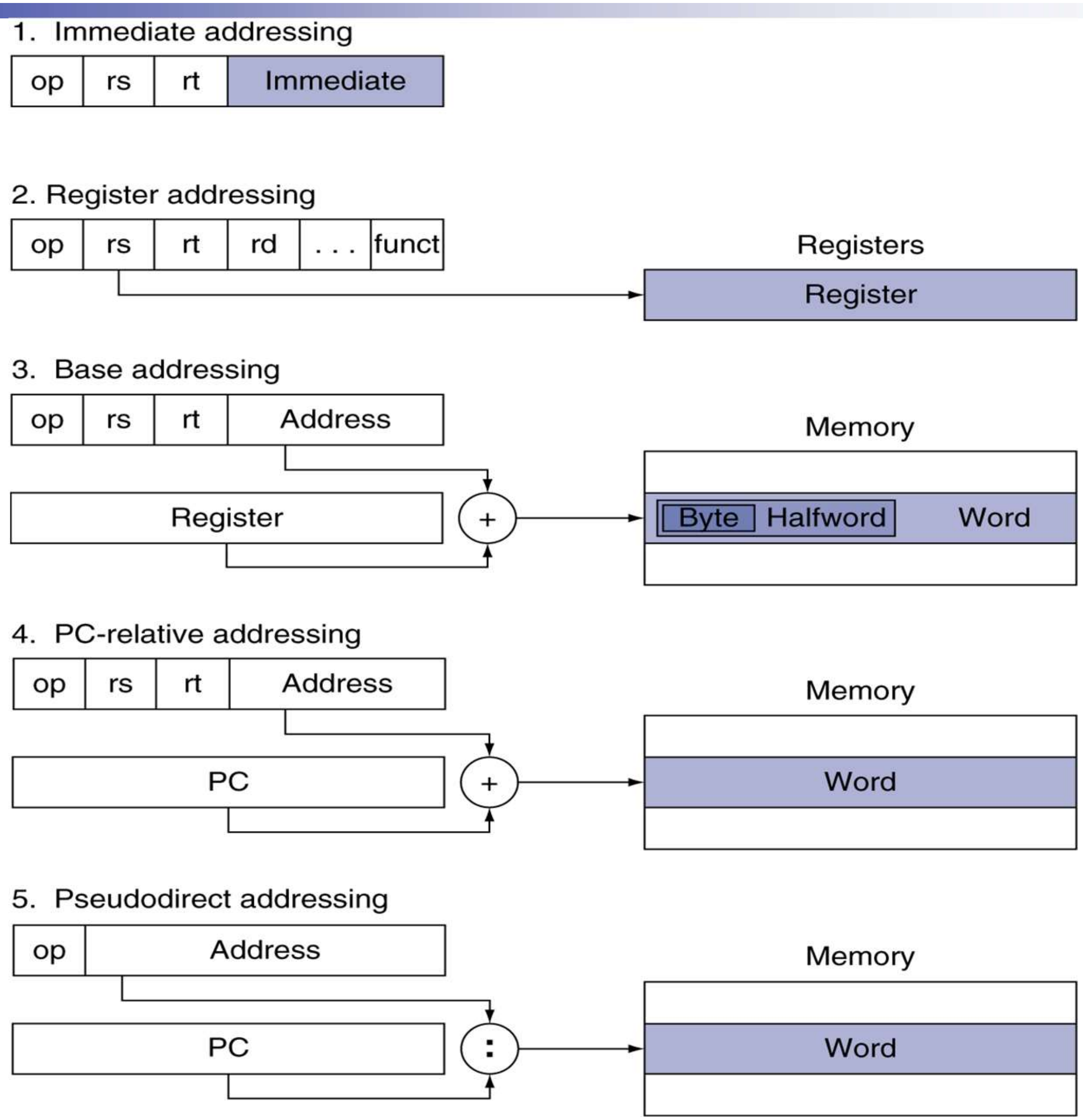
- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address $\times$ 4)

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example: how can we increase branching distance for the following instruction?

```
          beq $s0,$s1, L1
                 //(如果相等，则跳转，但是L1超出了16位地址范围)
                    ↓
          bne $s0,$s1, L2  //可以转换为无条件跳转
          j L1
     L2:  …
```

Addressing Mode Summary



主观题 10分

设置

如果目标指令地址与当前跳转指令地址之间的间隔超过了2^28，那么请问编译器如何来解决该问题?

正常使用主观题需2.0以上版本雨课堂

作答

Synchronization

- 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 and ISA 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

Motivation for Atomic Swap

- Build lock for multi-processor system
  - To govern access to some critical resource
  - 0 indicates lock is free; 1 indicates lock is unavailable; “lock” stored in memory
  - Processor tries to set the lock by doing an exchange of 1 which is in a register
  - Value returned from exchange is
    - 1 if another processor already acquired the lock;
    - 0 otherwise; in this case value changed to 1 to prevent another processor from retrieving 0
  - Locks allow breaking race conditions

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs); rt->(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 相当于加锁机制
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

- ll, sc can be used to build other synchronization primitives: atomic compare and swap, fetch and increment, etc.

主观题　2分

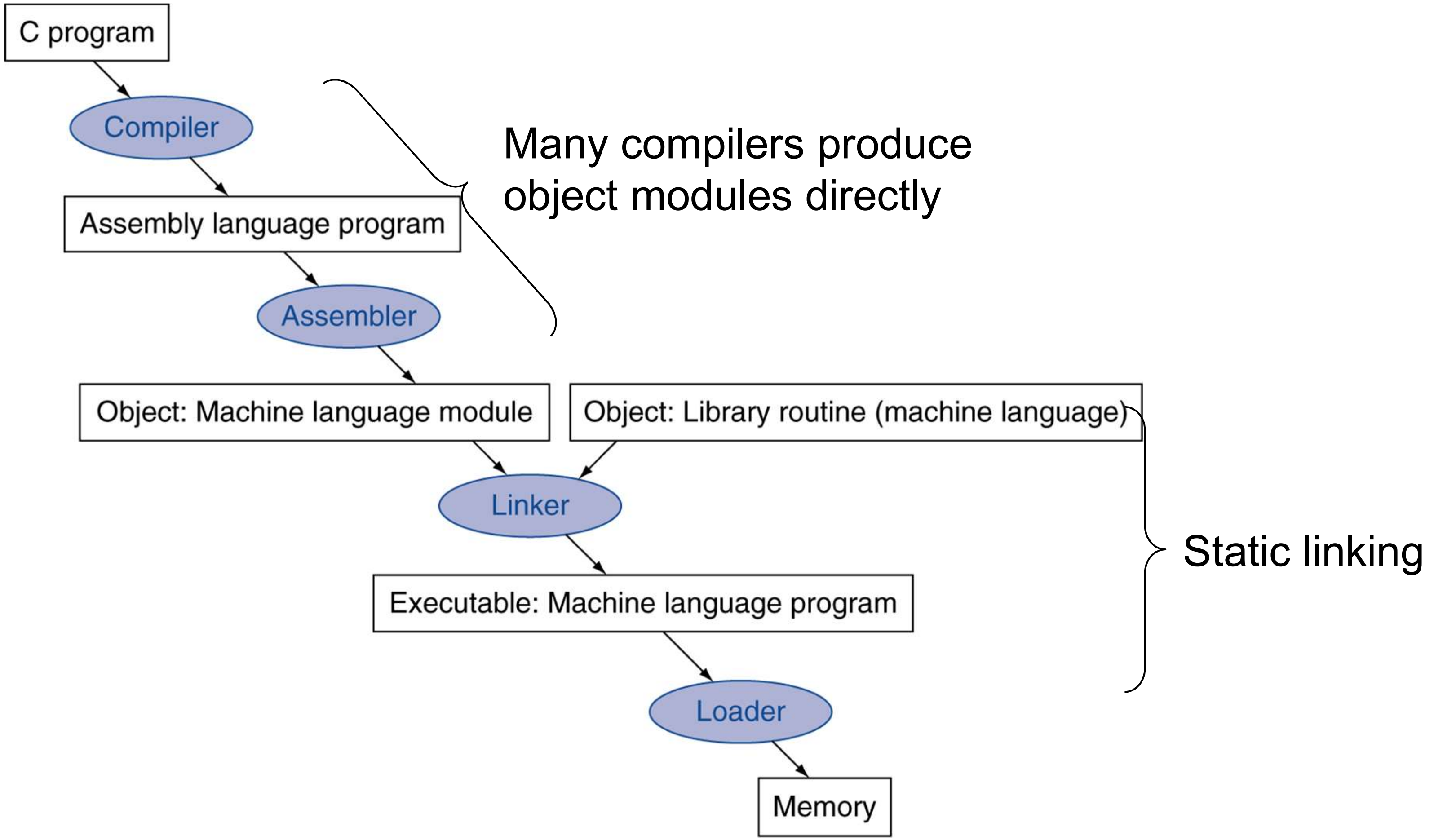
设置

请利用MIPS指令集中的LL 指令和 SC实现一个内存变量（内存地址为0($s1)）进行读取并+1的原子计数操作的代码。

正常使用主观题需2.0以上版本雨课堂

作答

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

http://www.chuquan.me/2018/05/21/elf-introduce/

Assembler Pseudo instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo instructions: 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

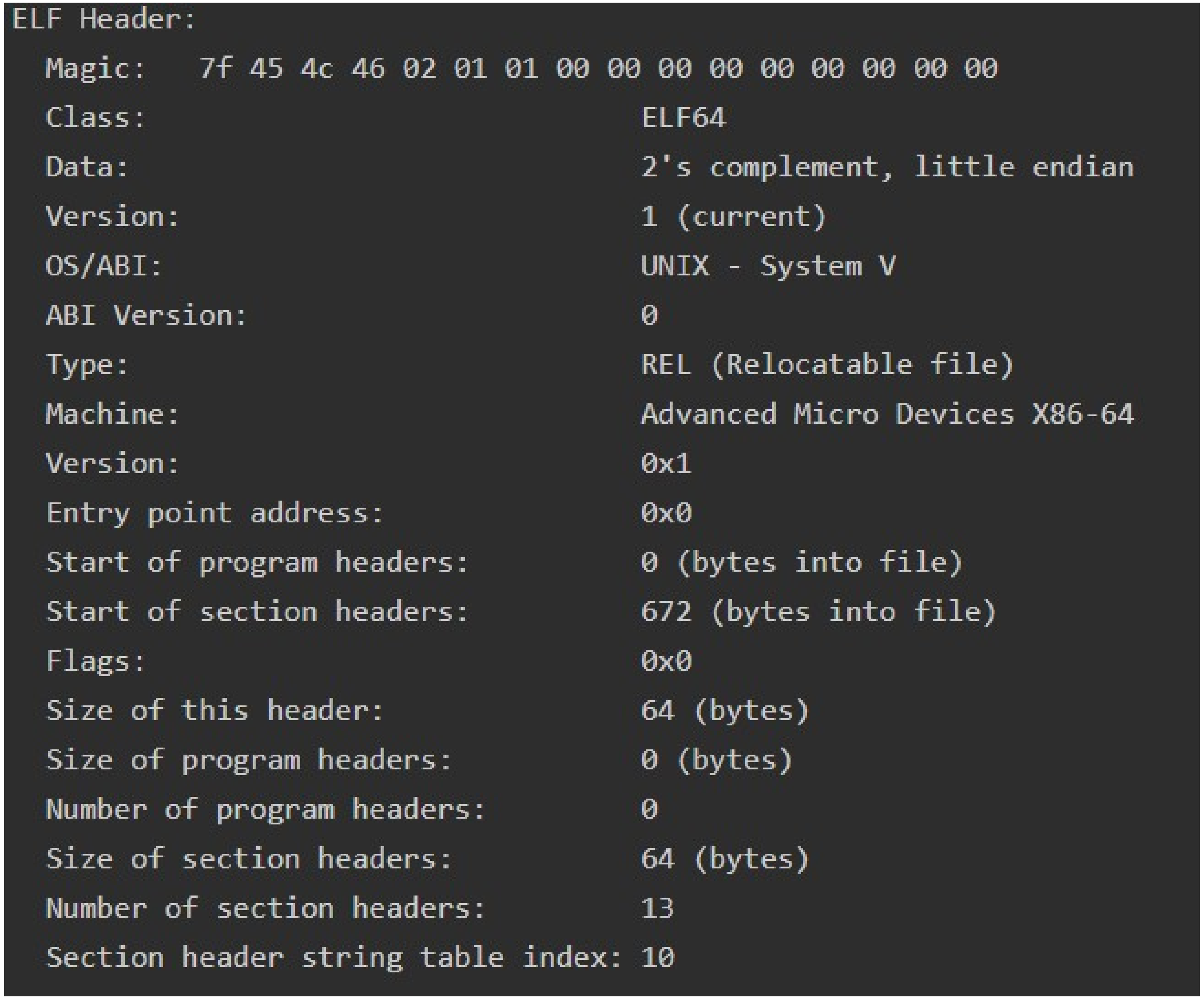
Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
- E.g. object file (*.o) for UNIX has 6 distinct parts:
  - 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 machine code with high level source code by debuggers

Linking Object Modules

- Produces an executable image
  1. Merges segments/object files/procedures
     - Rather than compiling/assembling whole program when a single line of a procedure is changed, procedures are compiled/assembled independently
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

Introduction of ELF



```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          672 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

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

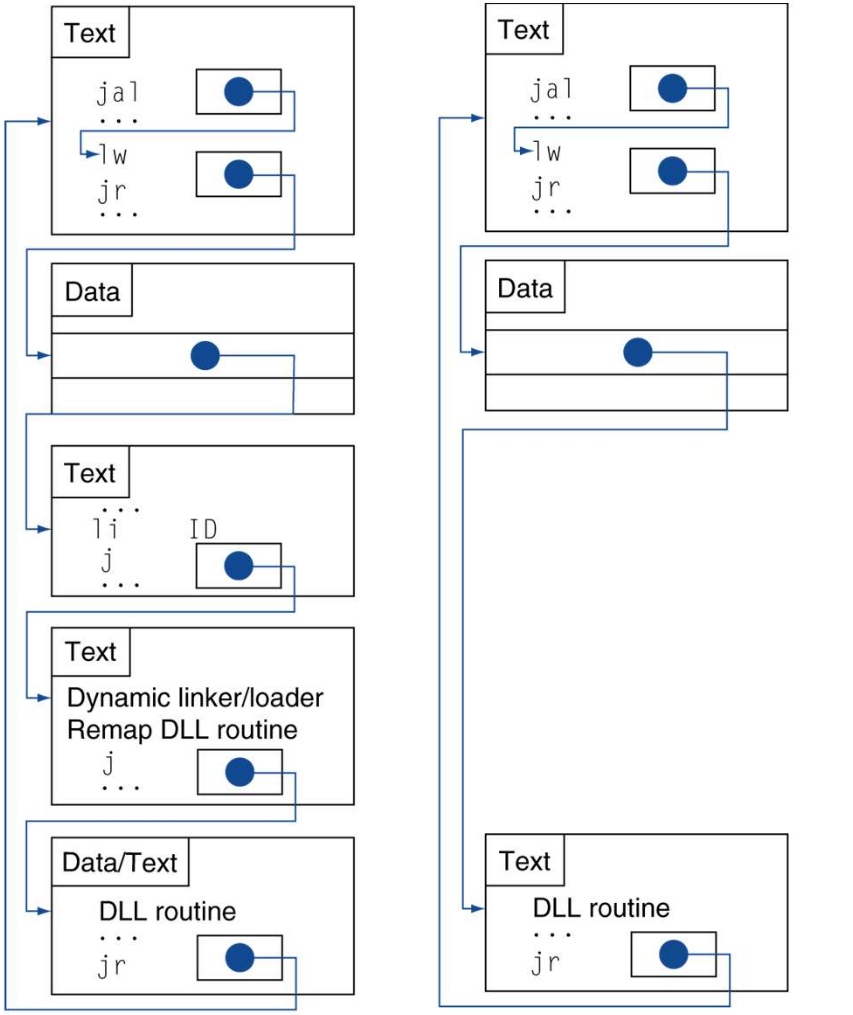
Static and Dynamic Linking

- So far discussed traditional approach to linking libraries before program executes
  - Static linking
    - If new version of a library released to fix bugs or support new hardware, statically linked program keeps using old version
    - All routines in a library are loaded, even if they are not used
- Dynamic linking: Do not link/load library routines until program is run
  - Requires procedure code to be relocatable
  - Automatically picks up new library versions
  - Dynamically linked libraries (DLLs)

Lazy Linkage

Routine linked only after it is called

Avoids image bloat caused by static linking of all referenced libraries



a. First call to DLL routine

b. Subsequent calls to DLL routine

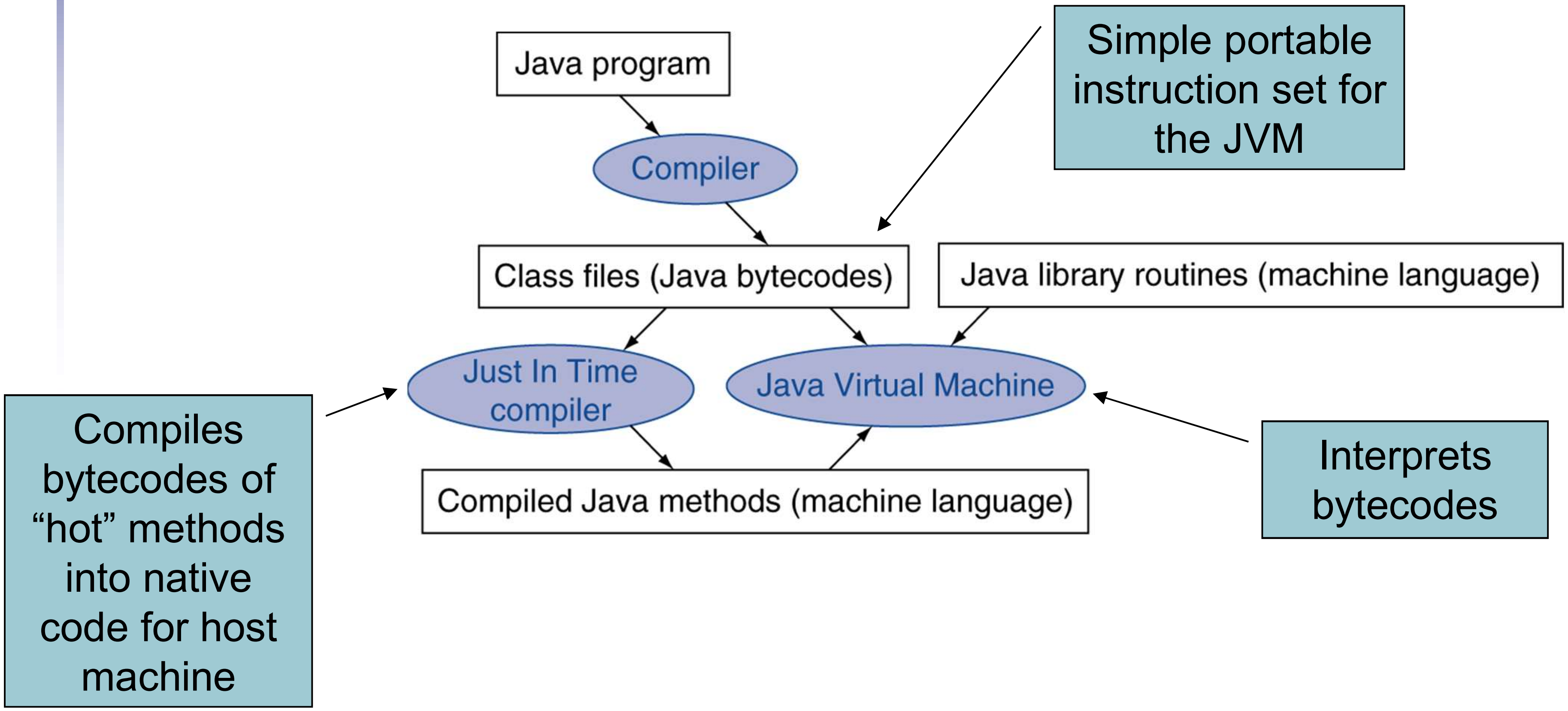
Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code

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
```

- Forgetting that sequential word addresses differ by 4 instead of 1 is a common mistake in assembly language programming

The Sort Procedure in C

- Non-leaf bubble/exchange sort (calls swap)

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

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

The Procedure Body

Move params

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

Outer loop

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

Inner 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
```

Pass params & call

```
         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
```

Inner loop

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

Outer loop

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

Mars MIPS汇编仿真工具的使用

- Missouri State University 开源的MIPS汇编仿真工具
- 具体的内容参考网址

http://courses.missouristate.edu/kenvollmar/mars/

Mars特性说明

1. 实现MIPS 155条基本指令和370条伪指令 17 系统函数（syscal）实现了大多数的文件和控制台输入/输出， 22个系统函数实现 多媒体输出以及随机数的生成等函数。

2 可以对汇编调试和跟踪程序执行

3 生成代码对应的机器指令

4 利用扩展的工具实现以下功能

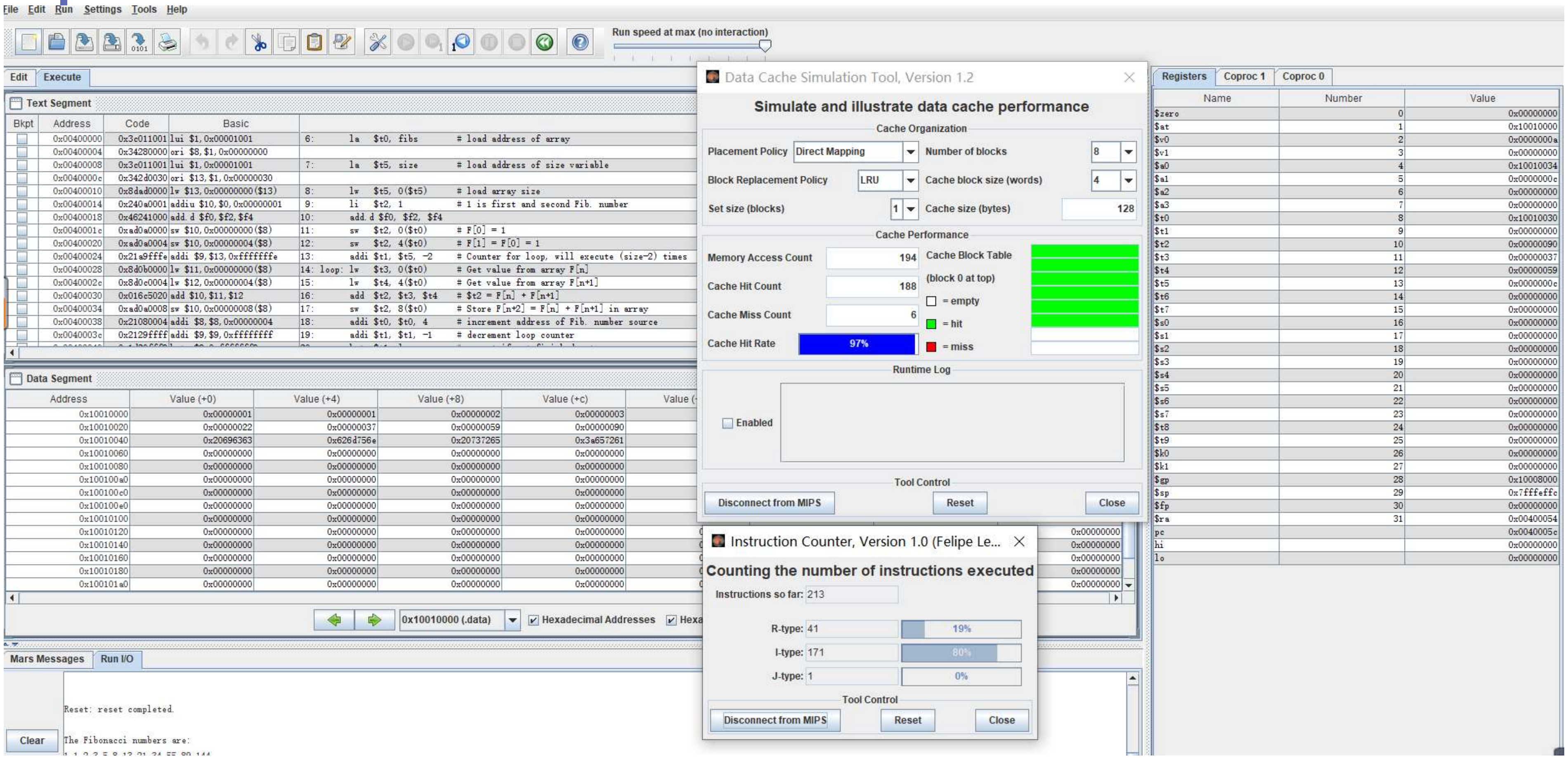
1） 观察不同 Cache管理的策略下Cache的命中率

2） 观察内存的热点区域

3） 观察不同分支预测策略的结果

4） 观察指令在简单的数据通路上的执行情况

Mars运行界面



Text Segment

Bkpt	Address	Code	Basic	
□	0x00400000	0x3c011001	lui $1, 0x00001001	6: la $t0, fibs # load address of array
□	0x00400004	0x34280000	ori $8, $1, 0x00000000	
□	0x00400008	0x3c011001	lui $1, 0x00001001	7: la $t5, size # load address of size variable
□	0x0040000c	0x342d0030	ori $13, $1, 0x00000030	
□	0x00400010	0x8dad0000	lw $13, 0x00000000($13)	8: lw $t5, 0($t5) # load array size
□	0x00400014	0x240a0001	addiu $10, $0, 0x00000001	9: li $t2, 1 # 1 is first and second Fib. number
□	0x00400018	0x46241000	add.d $f0, $f2, $f4	10: add.d $f0, $f2, $f4
□	0x0040001c	0xad0a0000	sw $10, 0x00000000($8)	11: sw $t2, 0($t0) # F[0] = 1
□	0x00400020	0xad0a0004	sw $10, 0x00000004($8)	12: sw $t2, 4($t0) # F[1] = F[0] = 1
□	0x00400024	0x21a9fffe	addi $9, $13, 0xfffffffe	13: addi $t1, $t5, -2 # Counter for loop, will execute (size-2) times
□	0x00400028	0x8d0b0000	lw $11, 0x00000000($8)	14: loop: lw $t3, 0($t0) # Get value from array F[n]
□	0x0040002c	0x8d0c0004	lw $12, 0x00000004($8)	15: lw $t4, 4($t0) # Get value from array F[n+1]
□	0x00400030	0x016c5020	add $10, $11, $12	16: add $t2, $t3, $t4 # $t2 = F[n] + F[n+1]
□	0x00400034	0xad0a0008	sw $10, 0x00000008($8)	17: sw $t2, 8($t0) # Store F[n+2] = F[n] + F[n+1] in array
□	0x00400038	0x21080004	addi $8, $8, 0x00000004	18: addi $t0, $t0, 4 # increment address of Fib. number source
□	0x0040003c	0x2129ffff	addi $9, $9, 0xffffffff	19: addi $t1, $t1, -1 # decrement loop counter

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00000001	0x00000001	0x00000002	0x00000003
0x10010020	0x00000022	0x00000037	0x00000059	0x00000090
0x10010040	0x20696363	0x626d756e	0x20737265	0x3a657261
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexa

Data Cache Simulation Tool, Version 1.2

Simulate and illustrate data cache performance

Cache Organization

Placement Policy: Direct Mapping
Number of blocks: 8
Block Replacement Policy: LRU
Cache block size (words): 4
Set size (blocks): 1
Cache size (bytes): 128

Cache Performance

Memory Access Count: 194
Cache Hit Count: 188
Cache Miss Count: 6
Cache Hit Rate: 97%

Cache Block Table (block 0 at top)
□ = empty
🟩 = hit
🟥 = miss

Runtime Log
□ Enabled

Tool Control
Disconnect from MIPS | Reset | Close

Instruction Counter, Version 1.0 (Felipe Le...

Counting the number of instructions executed

Instructions so far: 213

R-type: 41 19%
I-type: 171 80%
J-type: 1 0%

Tool Control
Disconnect from MIPS | Reset | Close

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x10010034
$a1	5	0x0000000c
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x10010030
$t1	9	0x00000000
$t2	10	0x00000090
$t3	11	0x00000037
$t4	12	0x00000059
$t5	13	0x0000000c
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400054
pc		0x0040005c
hi		0x00000000
lo		0x00000000

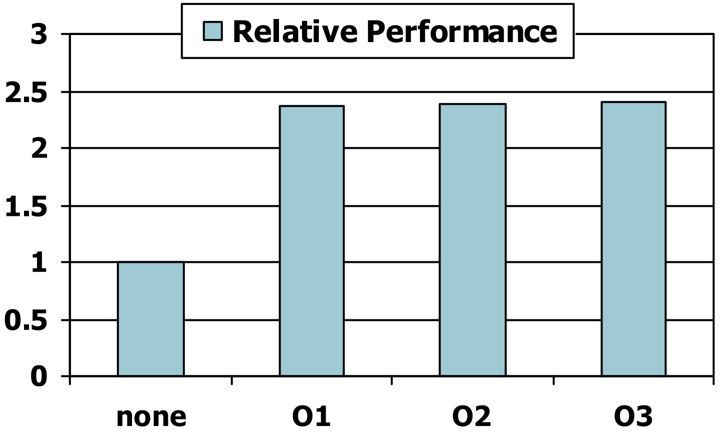
```
Reset: reset completed.

The Fibonacci numbers are:
1 1 2 3 5 8 13 21 34 55 89 144
```

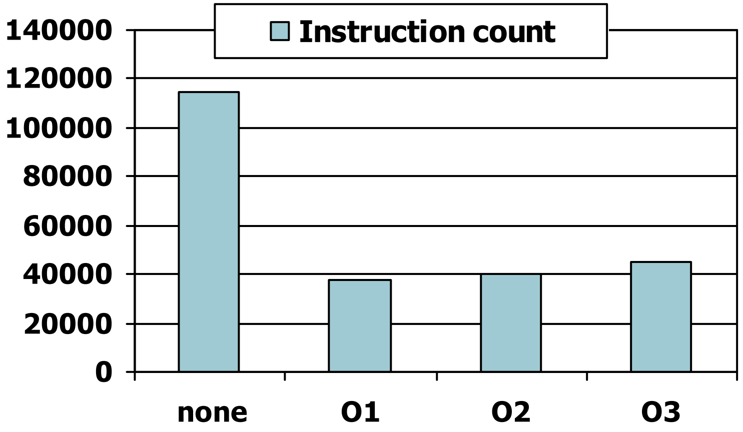
Effect of Compiler & Language & Algorithm

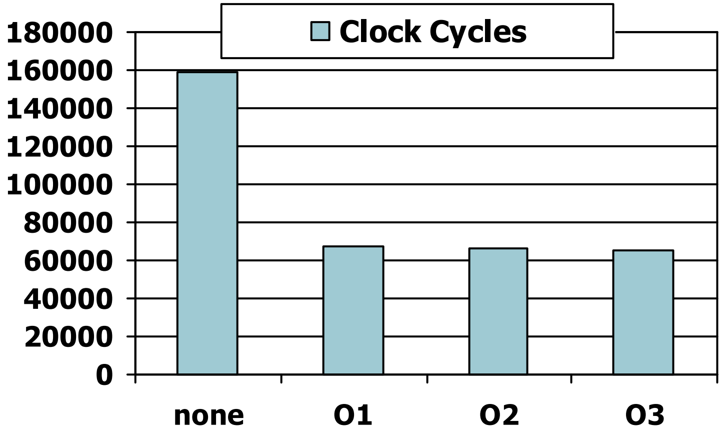
	Instruction_count	CPI	clock_cycle
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	X
Core organization		X	X
Technology			X

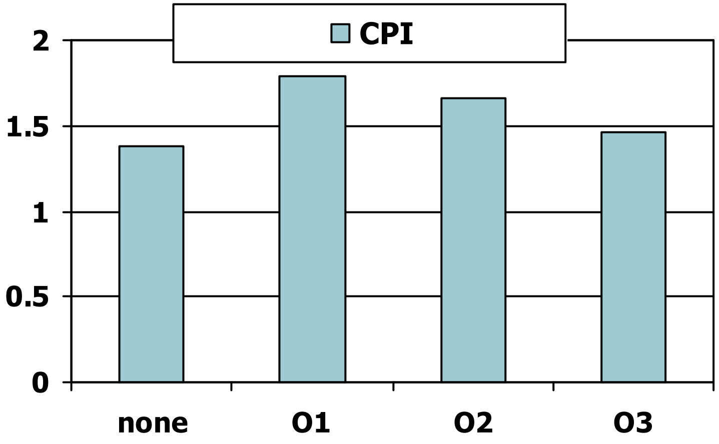
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux



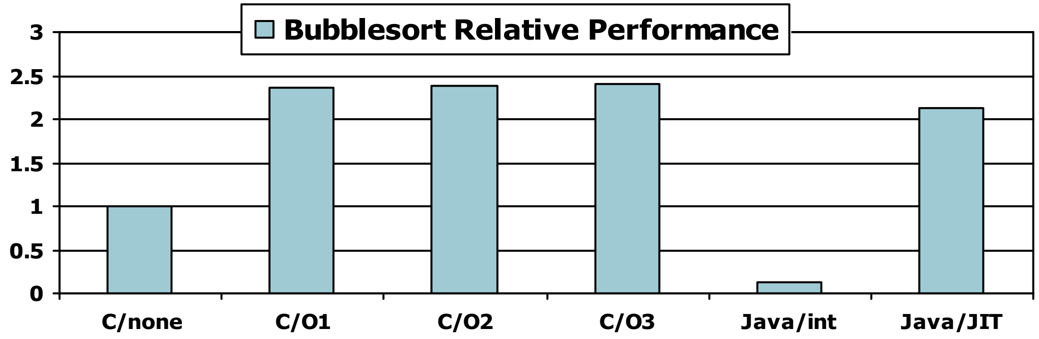


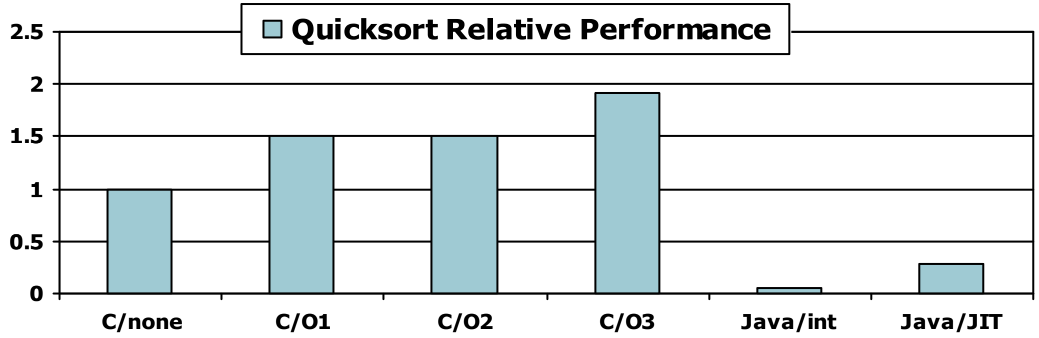


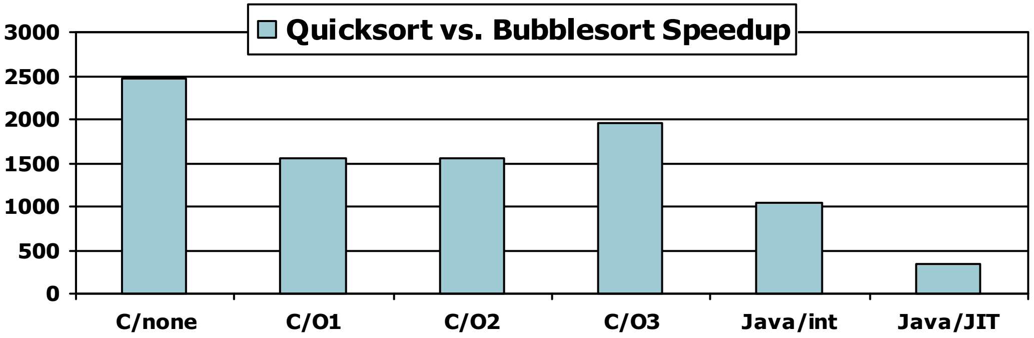


https://gcc.gnu.org/

Effect 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

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

Example: Clearing an Array

```
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
```

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
    - Avoid using pointers!!

主观题 10分

设置

课后练习题

1. 请研究一下Linux和Windows可执行程序的基本结构

2. 请在Linux系统中使用MIPS-GCC工具链，使用C语言写一个简单输出“hello world!”的程序，并编译为ELF格式的可执行程序

3. 观察汇编代码的格式，分析可执行程序的基本结构；

4. 将编译生成的逻辑程序在MARS仿真环境下运行。

5. 撰写分析报告并提交到Sakai中，分析报告的文档格式自拟。

正常使用主观题需2.0以上版本雨课堂

作答

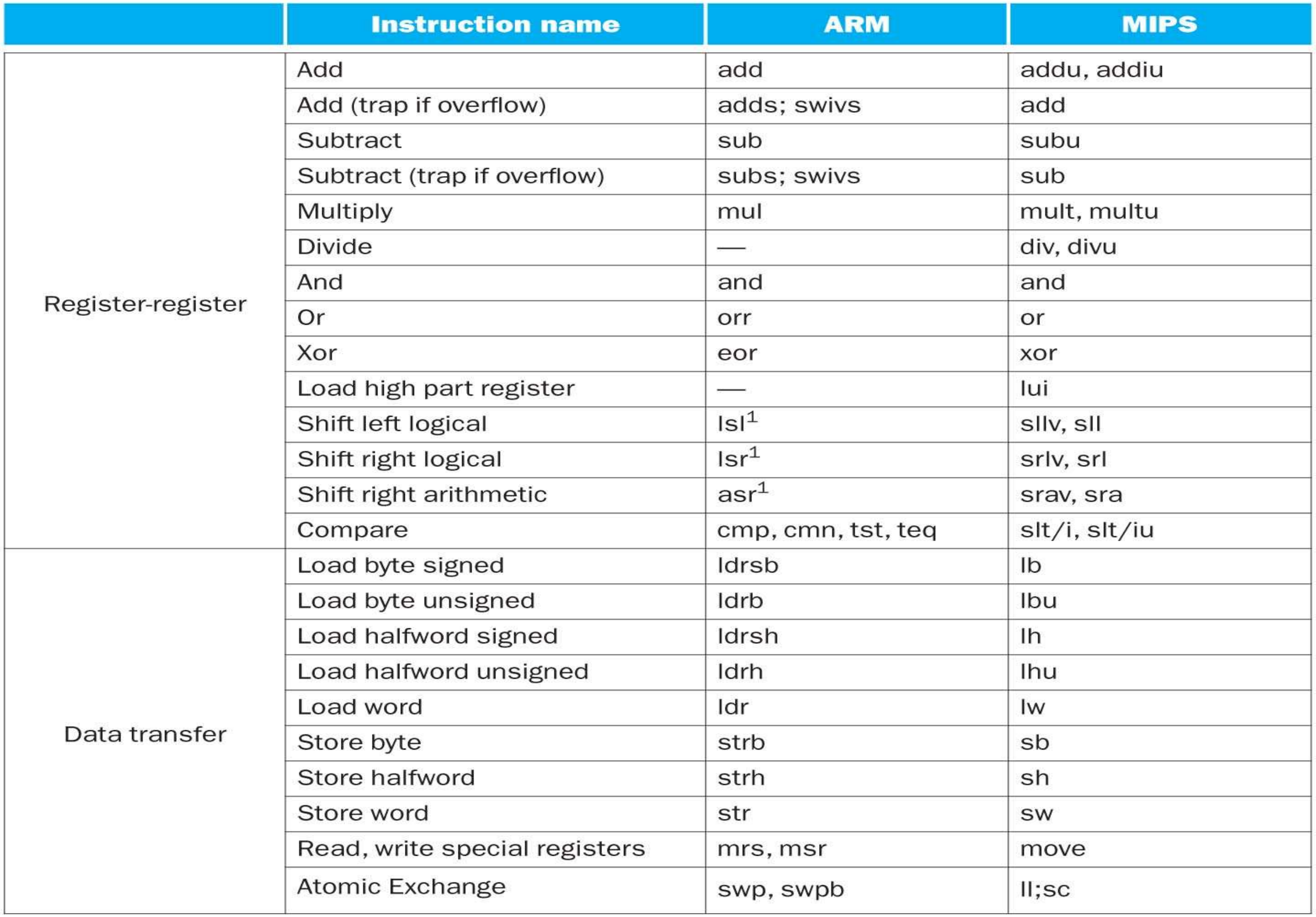
ARM & MIPS Similarities

- 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

§ 2.16 Real Stuff: ARM Instructions

ARM & MIPS Similarities

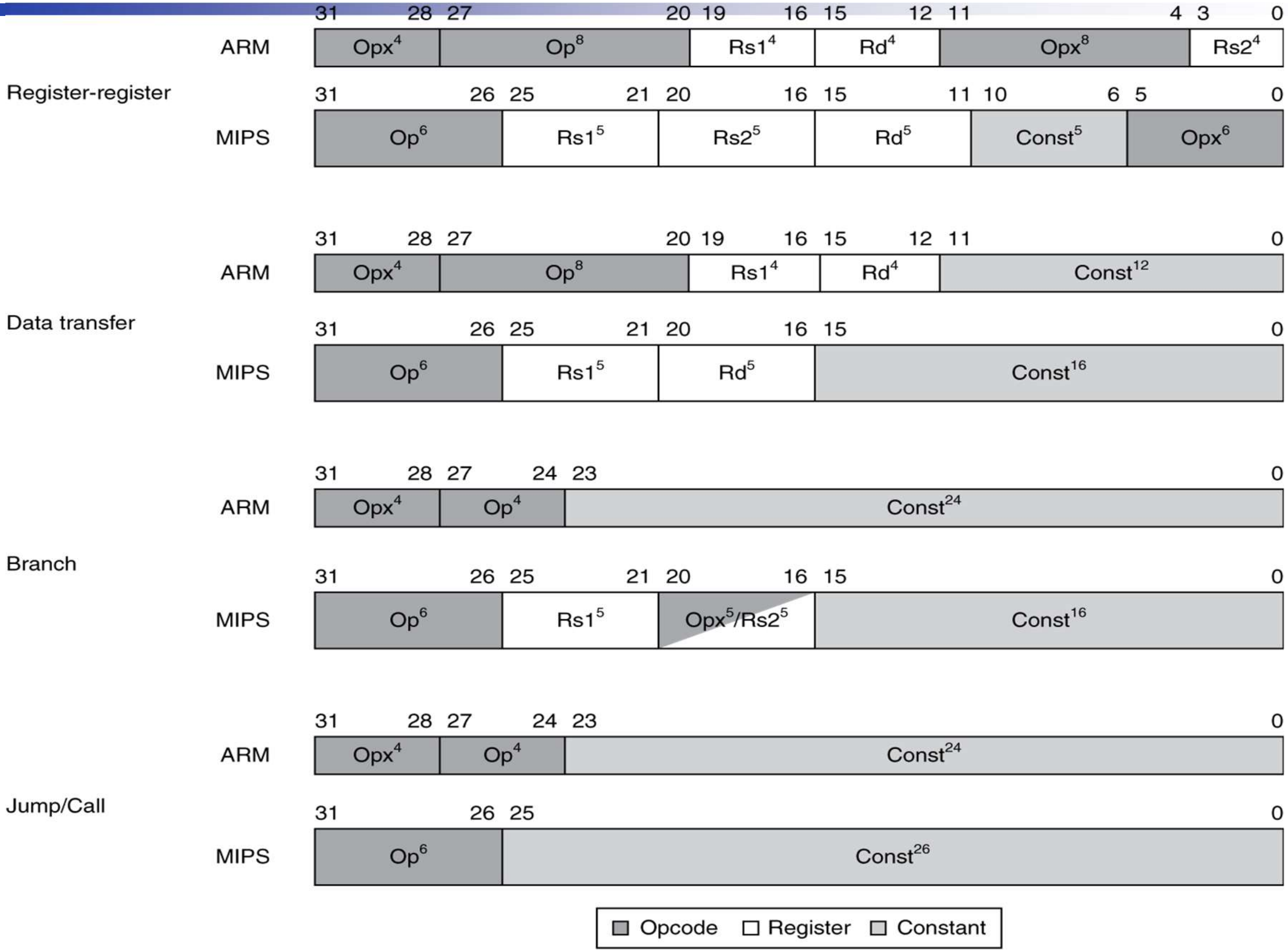


	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl[1]	sllv, sll
	Shift right logical	lsr[1]	srlv, srl
	Shift right arithmetic	asr[1]	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i, slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

Compare and Branch in ARM

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

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 | Opx^5/Rs2^5 | Const^{16}

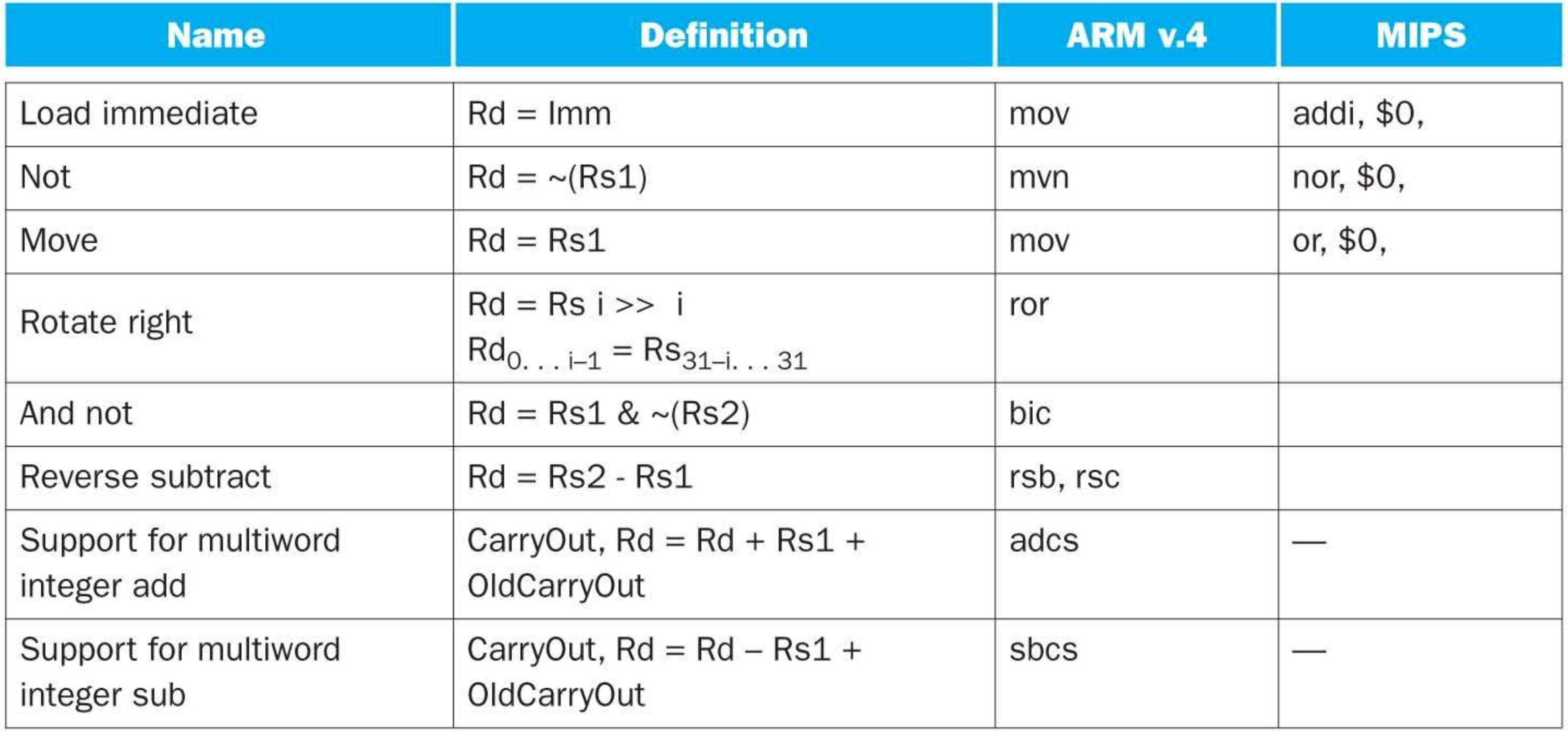
Jump/Call

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

Unique ARM Instructions



Name	Definition	ARM v.4	MIPS
Load immediate	Rd = Imm	mov	addi, $0,
Not	Rd = ~(Rs1)	mvn	nor, $0,
Move	Rd = Rs1	mov	or, $0,
Rotate right	Rd = Rs i >> i $Rd_{0...i-1} = Rs_{31-i...31}$	ror	
And not	Rd = Rs1 & ~(Rs2)	bic	
Reverse subtract	Rd = Rs2 - Rs1	rsb, rsc	
Support for multiword integer add	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
Support for multiword integer sub	CarryOut, Rd = Rd – Rs1 + OldCarryOut	sbcs	—

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

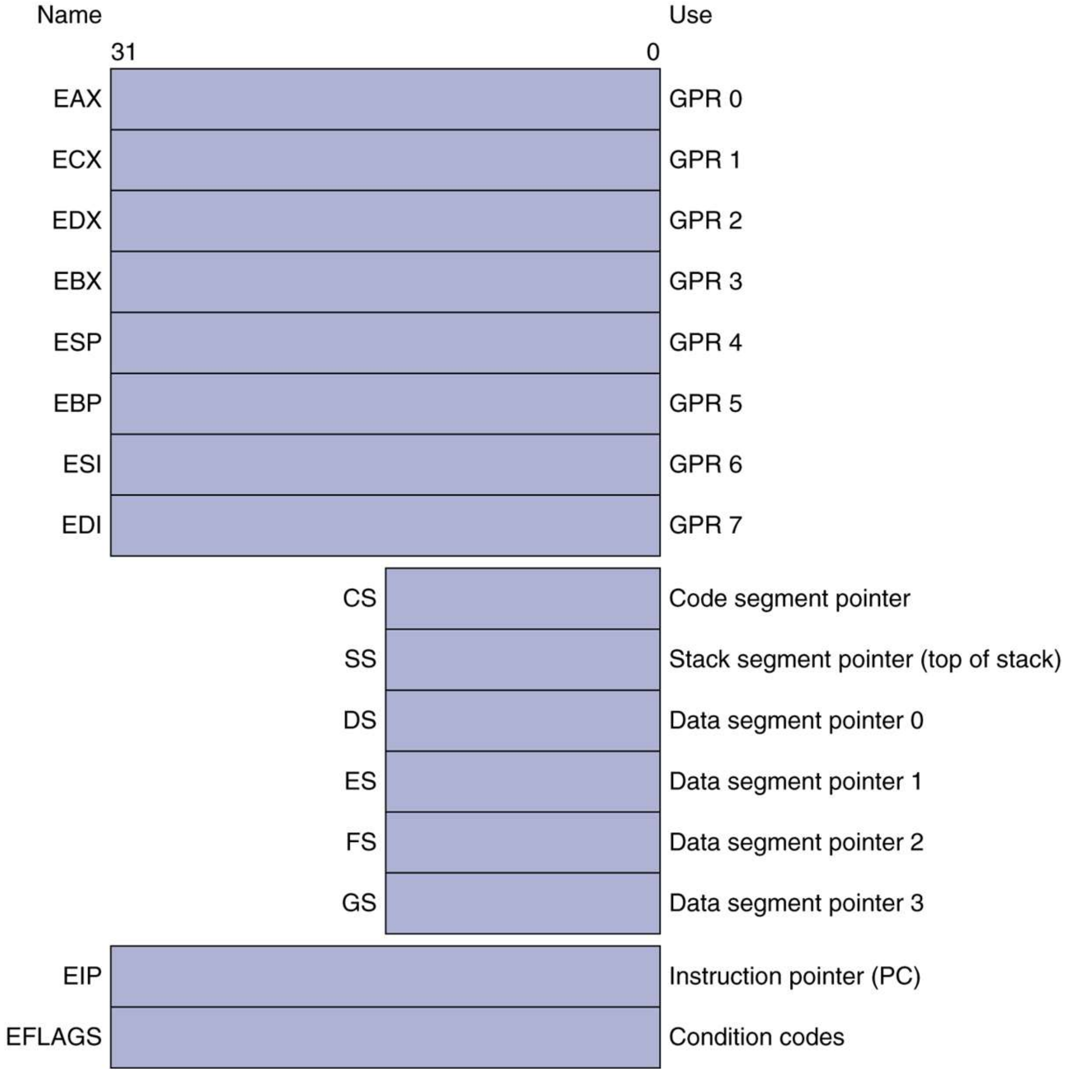
The Intel x86 ISA

- 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 $500M 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 144 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
- Existing x86 software base at each step too important to jeopardize with significant architectural changes
  - x86 extended by one instruction per month since inception!
  - is an architecture that is difficult to explain and hard to love
  - despite lacking "technical elegance", x86 family pervasive today

Basic x86 Registers (80386)



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

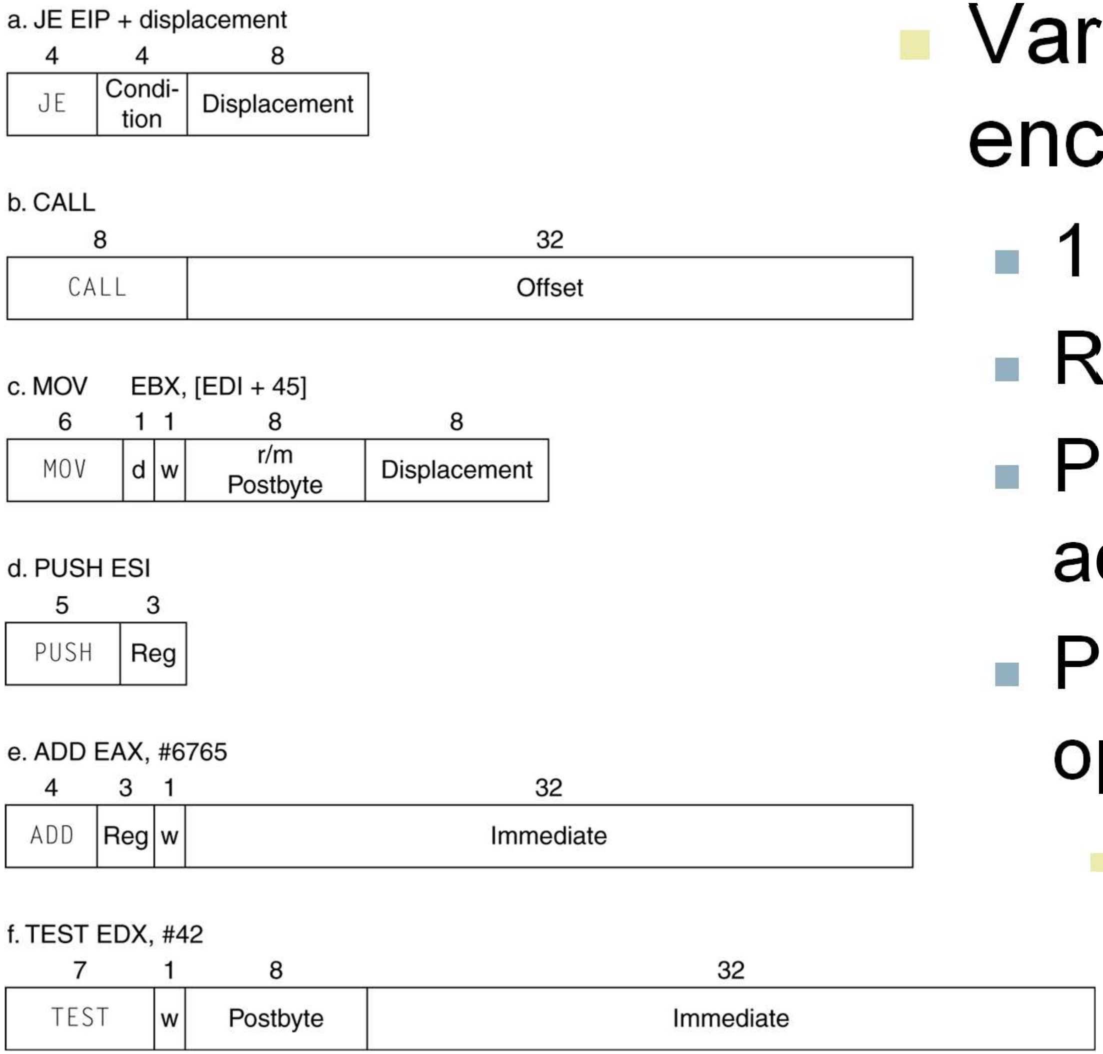
Basic x86 Addressing Modes

- 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



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

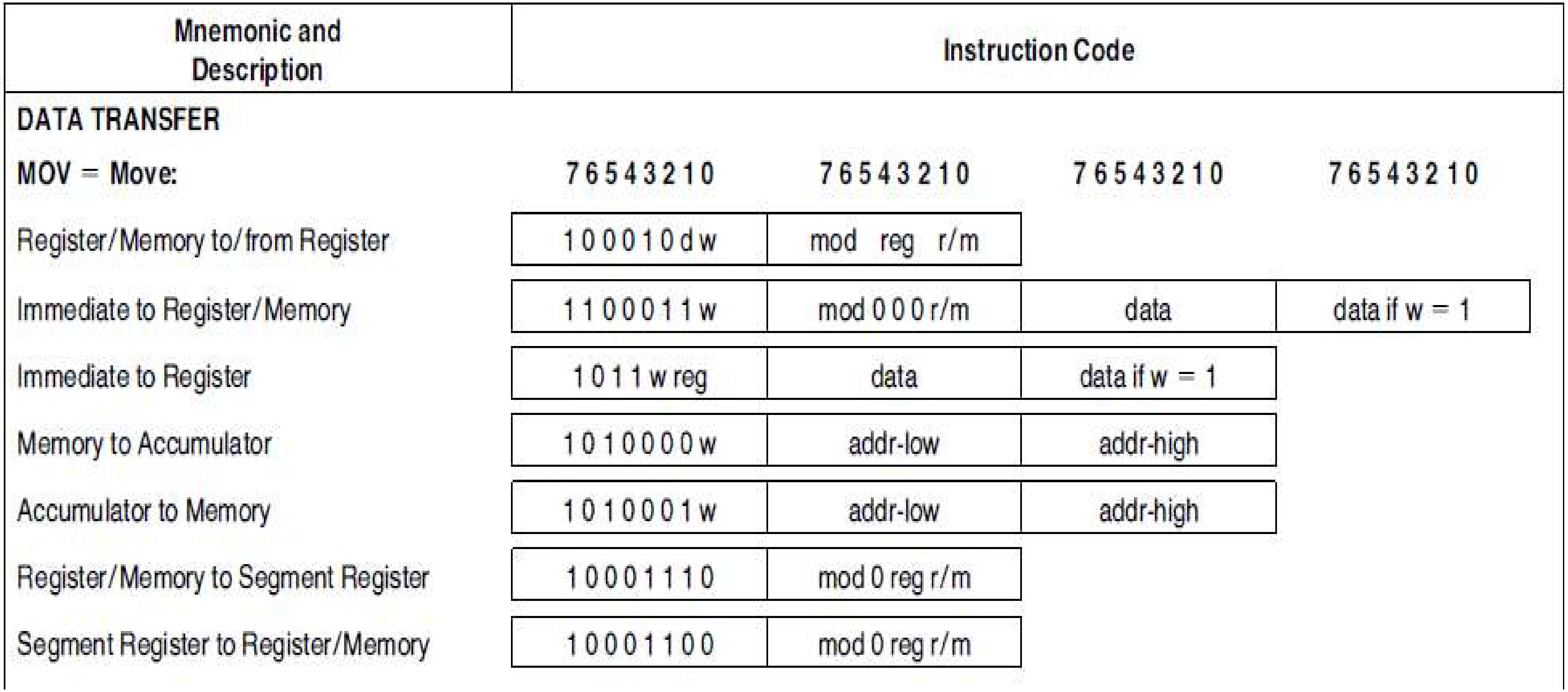
f. TEST EDX, #42

7 1 8 32

TEST | w | Postbyte | Immediate

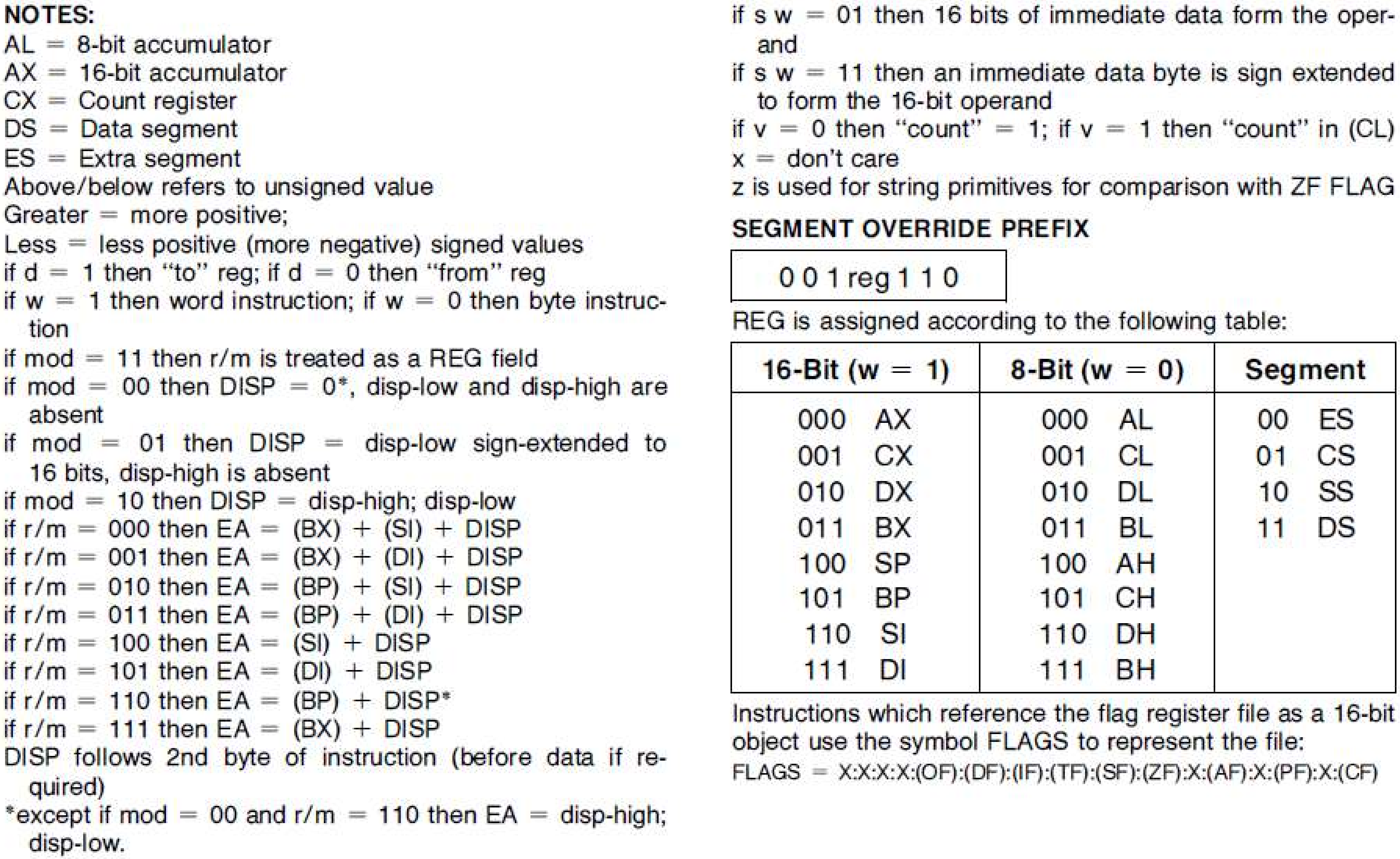
- Variable length encoding
  - 1 to 17 bytes
  - Rightly called **CISC**
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Intel 80x86指令的编码举例（MOV）



Mnemonic and Description	Instruction Code			
DATA TRANSFER				
MOV = Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		

X86指令的编码的说明



NOTES:
AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction
if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)
*except if mod = 00 and r/m = 110 then EA = disp-high; disp-low.

if s w = 01 then 16 bits of immediate data form the operand
if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with ZF FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-Bit (w = 1)		8-Bit (w = 0)		Segment	
000	AX	000	AL	00	ES
001	CX	001	CL	01	CS
010	DX	010	DL	10	SS
011	BX	011	BL	11	DS
100	SP	100	AH		
101	BP	101	CH		
110	SI	110	DH		
111	DI	111	BH		

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:
FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler micro-operations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
- Comparable performance to RISC
  - Compilers avoid complex instructions

开源指令集架构：RISC-V

RISC-V的基金会则是由Google, Microsoft, Qualcomm, AMD和华为等六十个会员组成，他们共同推进架构的发展。

Nvidia也打算将这个微控制器类型的Risc-V设计到其芯片里面。而类似谷歌这些开发内部使用芯片的公司，则打算将RISC-V“调教”到使用于服务器。这可以帮助他们减少购买Intel昂贵服务器芯片的次数。

主要的特点:模块化

RISC-V架构相比其他成熟的商业架构的最大一个不同还在于它是一个模块化的架构，通过一套统一的架构满足各种不同的应用。这种模块化是x86与ARM架构所不具备的。

以ARM的架构为例，ARM的架构分为A、R和M三个系列，分别针对于Application（应用系统）、Real-Time（实时）和Embedded（嵌入式）三个领域，彼此之间并不兼容。

模块化的RISC-V架构能够使得用户能够灵活选择不同的模块组合，以满足不同的应用场景。针对于小面积低功耗嵌入式场景，用户可以选择RV32IC组合的指令集，仅使用Machine Mode（机器模式）；而高性能应用操作系统场景则可以选择譬如RV32IMFDC的指令集，使用Machine Mode（机器模式）与User Mode（用户模式）两种模式。

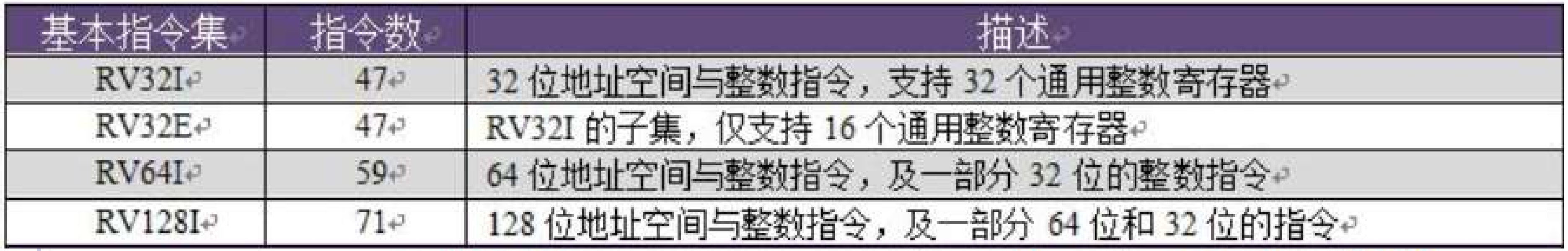
RISC-V 指令集架构简介（指令编码）

<table>
<tr><td>31</td><td>30 25</td><td>24 21</td><td>20</td><td>19 15</td><td>14 12</td><td>11 8</td><td>7</td><td>6 0</td><td></td></tr>
<tr><td colspan="2">funct7</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="4">imm[11:0]</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td>imm[12]</td><td>imm[10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td>imm[20]</td><td colspan="2">imm[10:1]</td><td>imm[11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td><td>J-type</td></tr>
</table>

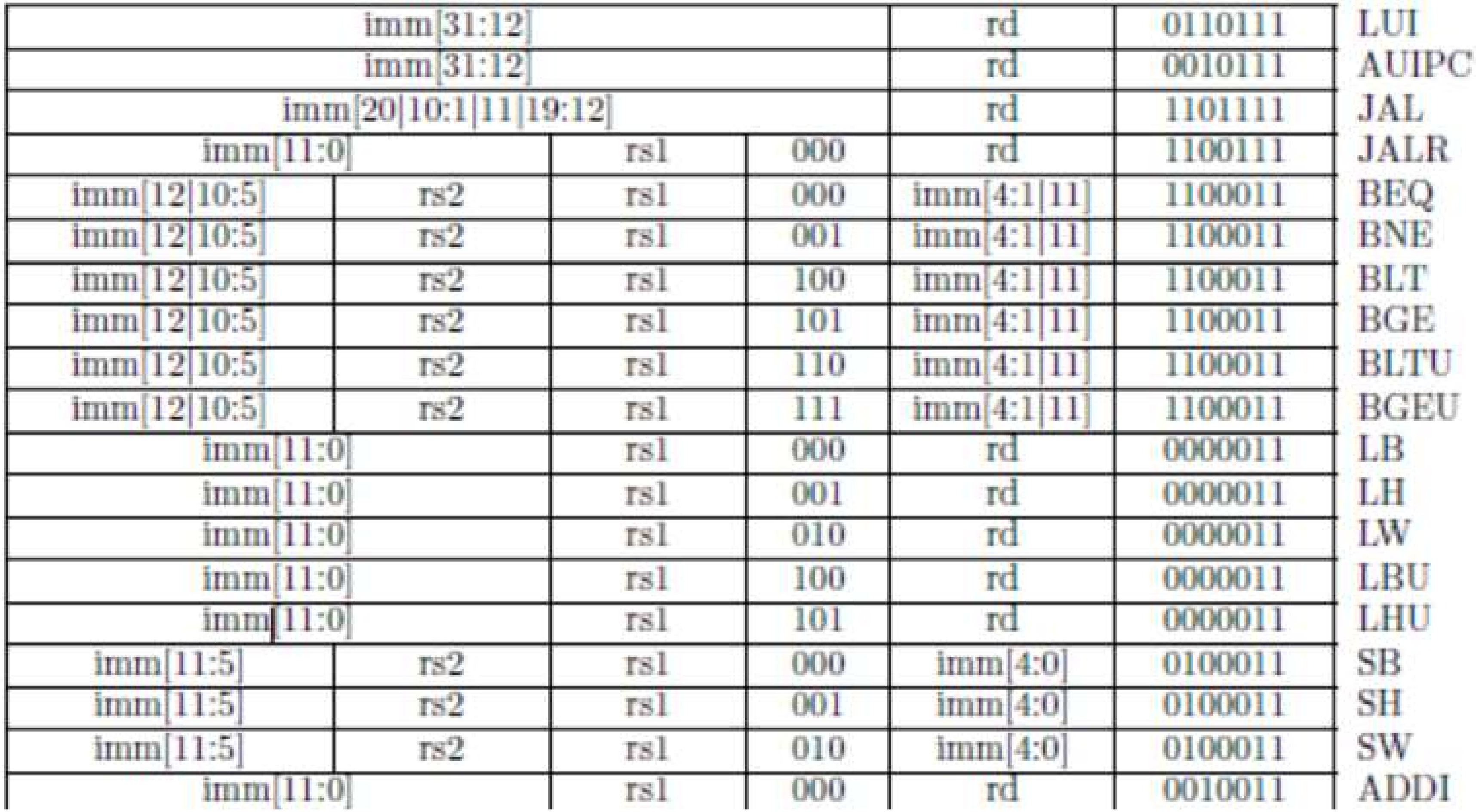
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

特点：极度精简



基本指令集	指令数	描述
RV32I	47	32 位地址空间与整数指令，支持 32 个通用整数寄存器
RV32E	47	RV32I 的子集，仅支持 16 个通用整数寄存器
RV64I	59	64 位地址空间与整数指令，及一部分 32 位的整数指令
RV128I	71	128 位地址空间与整数指令，及一部分 64 位和 32 位的指令



imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI

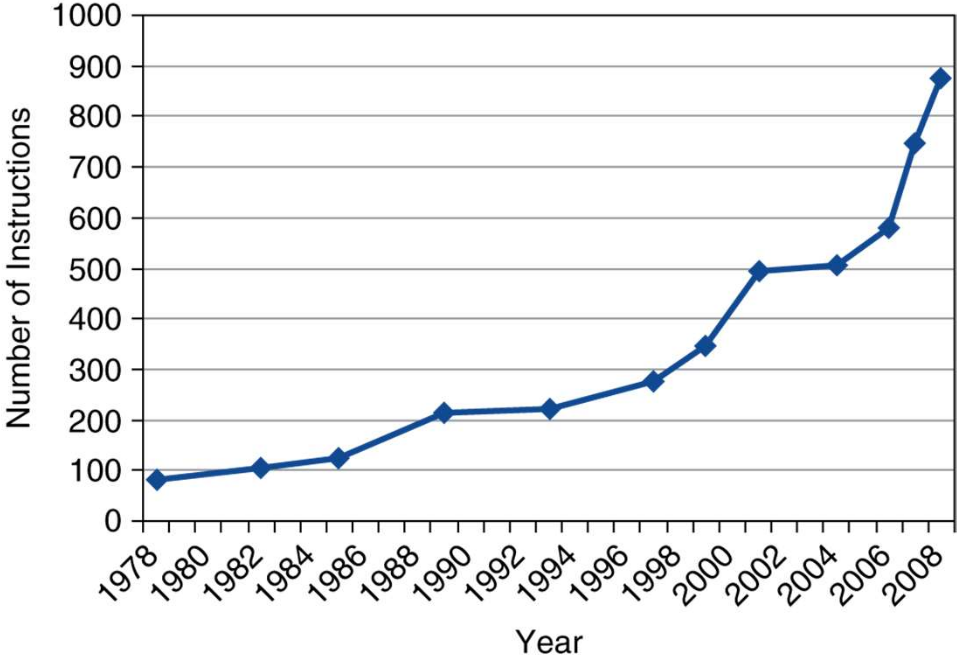
Fallacies

§ 2.18 Fallacies and Pitfalls

- 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 (as is the case for assembly code vs. C or Java) ⇒ more errors and less productivity

Fallacies

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



x86 instruction set

Pitfalls

- Forgetting that 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 to a local array/variable back via an argument
  - Pointer becomes invalid when stack popped

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

- Measured MIPS instruction execution frequencies in benchmark programs
  - Consider making the common case fast

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%

Review: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

Assignment 2

- Homework assignment

  2.1~2.5, 2.7, 2.10, 2.39, 2.40

- Read Chapter 2 of

Advanced Material: Compiling C and Interpreting Java