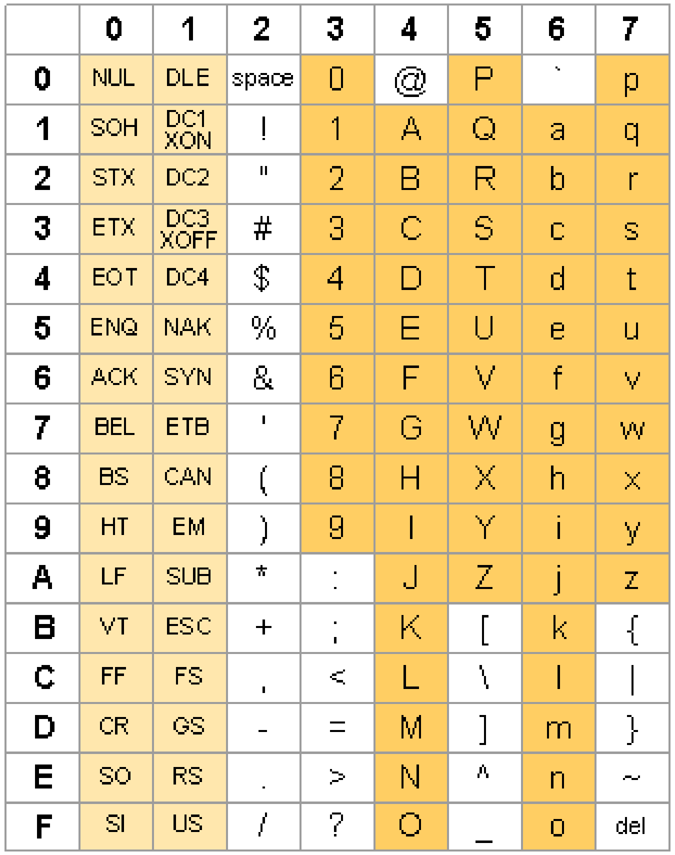
Character Data

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



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

Byte/Halfword Operations

- MIPS byte/halfword load/store are common for string processing
- `lb $t0, 0($sp)` `//load with sign extension`
  - Load a byte at the rs+offset address into rt
  - Sign extend to 32 bits in rt
- `Sb $t0, 0($sp)`
  - Store just rightmost byte

$t0=0x12345678

$t0=? After lb $t0 0($sp)

values starting from $sp

0x88 0x77 0x66 0x55

0xFFFFFF88

Byte/Halfword Operations

- `lbu $t0, 0($sp)`   `//load without sign extension`
  - Load a byte at the rs+offset address into rt
  - Sign bit is NOT extended in rt

- No `sbu` …

Other similar instructions

- `lh rt, offset(rs)    ; load half word`
  - Load a halfword at the rs+offset address into rt
  - Sign extend to 32 bits in rt
- `lhu rt, offset(rs) ;load half word unsigned`
- `sh rt, offset(rs) ; store halfword`
  - Store just rightmost halfword
- No `shu` …

String Copy Example

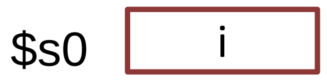
- C code (naïve):
  - Null-terminated string

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

  - Address of x in $a0, address of y in $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;
}
```



- 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, next byte
    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 range is $-2^{16} \sim 2^{16}-1$)
- Sometimes we need 32-bit constant, but an instruction can't have 32-bit constant (no space for op code)

  => combine lui (load upper imm.) and ori (or imm.) instructions to achieve this

  – lui rt, constant
    - Copies 16-bit constant to left 16 bits of rt
    - Clears right/lower 16 bits of rt to 0

Question: Steps to set \$s0 to 4,000,000

4000000_{10}=$0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_2$

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

Branch Addressing (for beq, bne)

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

Addr.	Inst.
8 (01000)	beq $t0 $t1 $\boxed{}$
12(01100)	….
16(10000)	…..
20(10100)	…..

- PC-relative addressing
  - Address is always a multiple of 4 => offset/4 is stored in the instruction
  - Target address = PC + (Address $\times$ 4)
  - Note, PC is already incremented by 4

op	rs	rt	Address (=Offset/4)
6 bits	5 bits	5 bits	16 bits

If the above beq go to address 20 when $t0==$t1, $\boxed{}$=? 2

000100	01000	01001	0000 0000 0000 0010
beq	$t0	$t1	

Example

- Suppose $s0=$t1,
- (1) find target address of beq instruction
- (2) the next instruction to be executed

addr field

Addr.	Inst.
4(00100)	beq $s0 $t1 2
8(01000)	Inst1……
12(01100)	Inst2…..
16(10000)	Inst3….
20(10100)	Inst4….

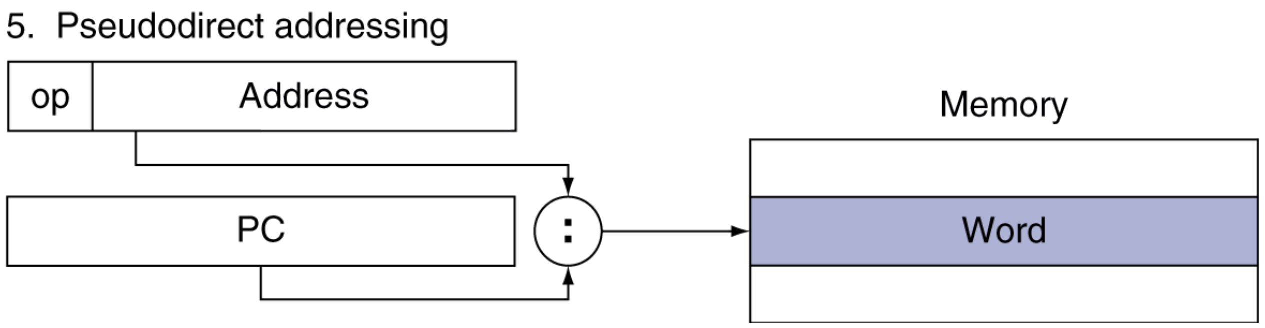
- Target address= 8 (i.e., PC) + 2*4=16
- Next instruction to be executed: inst3

Jump Addressing

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

op	address
6 bits	26 bits

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



Jump Example

- Assume PC=40000000_{16}, what is the target address of the jump instruction?

000010	00 00000000 00000010 00000001
6 bits	26 bits

Address in the instruction= 0x0000201

Target Address= PC[31:28]+0000201_{16}*4= 0x40000804

Target Addressing Example

- Loop code from earlier example (assume PC[31:28]=0000)
  - 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	(1)=???		
`      addi $s3, $s3, 1`	80016	8	19	19	1		
`      j    Loop`	80020	2	(2)=???				
`Exit: …`	80024						

(1) : 80016+ x*4 = 80024 , (1)=2

(2) : PC[31:28]+20000*4=80000 (2)= 20000

Branching Far Away

- If branch target is too far to be encoded with 16-bit offset, assembler insert an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump.
- Example

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

$\Rightarrow$ L1 can only be 16bit address

↓

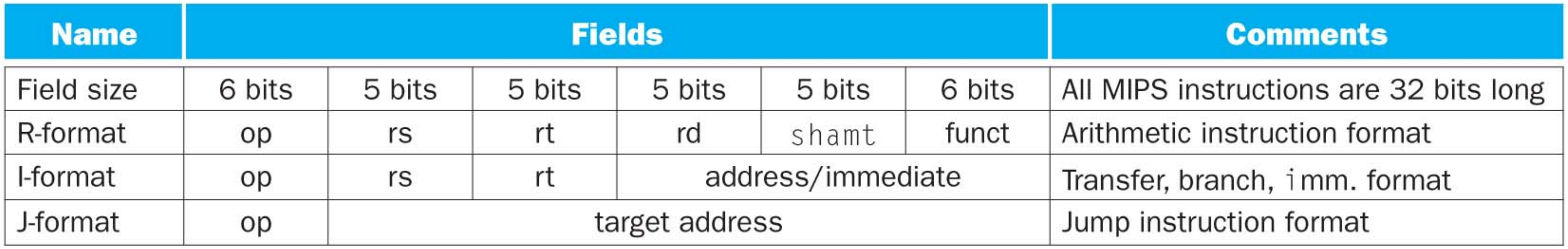
```
bne $s0,$s1, L2
j L1
L2:    …
```

$\Rightarrow$ L2 can only be 26bit

j can jump farer than beq

Summary: Instruction Format

- R-format: add, and, or …
- I-format: beq, bneq, addi, …
- J-format: j, jal

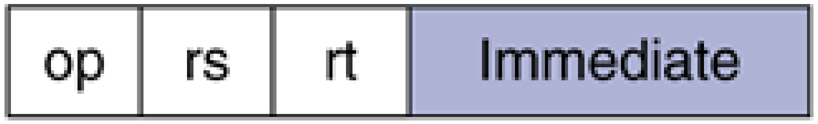


Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Summary: Addressing Mode

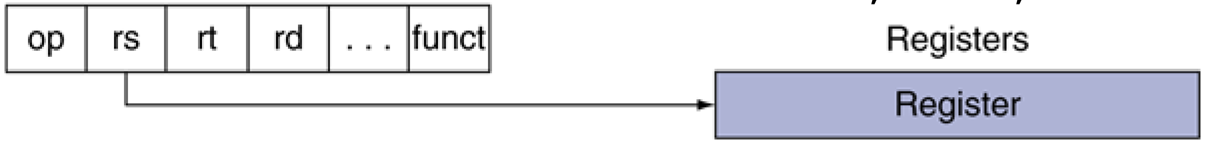
- Immediate addressing: operand is a constant within the instruction (e.g. addi)

```
addi $s1, $s0, 1   # s1 = s0+1
```



- Register addressing: operand is a register (e.g. add, nor)

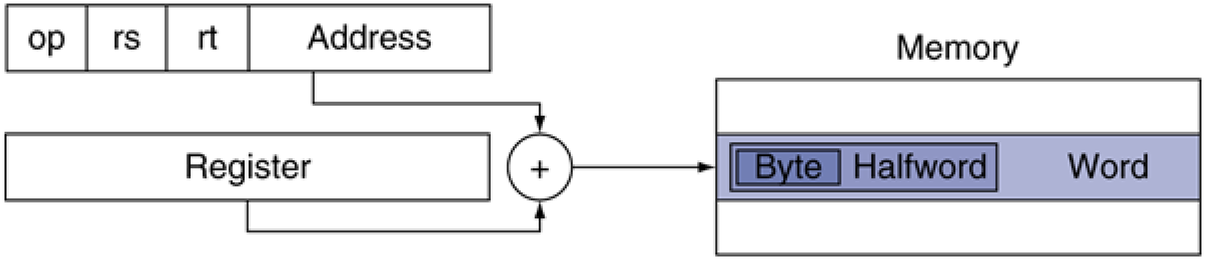
```
add $s1, $s0, $s2
```



- Base or displacement addressing: operand is at the memory location (e.g. lw, sw)

```
lw $t0, 32($s3)
```

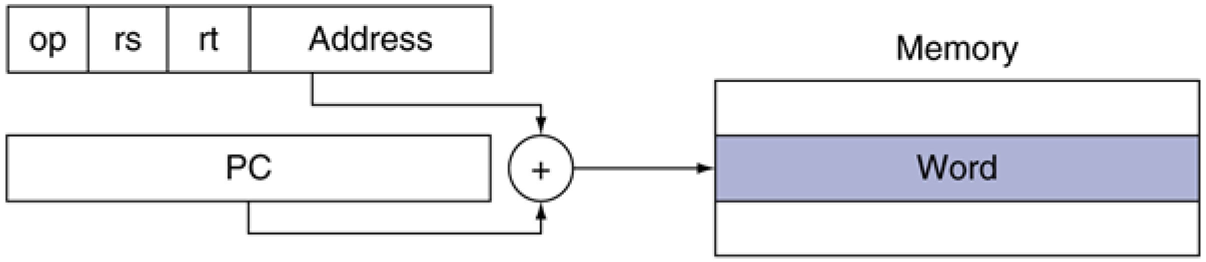
base



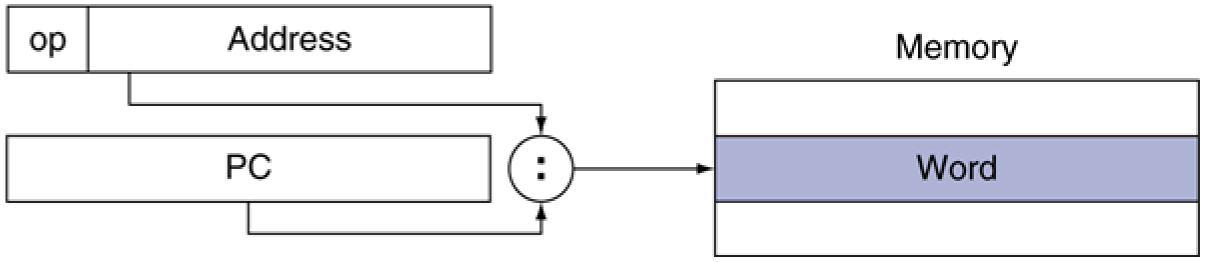
Summary: Addressing Mode (2)

- PC-relative addressing: branch address is the sum of 16-bit constant (shift left 2 bits) and PC (e.g. beq)

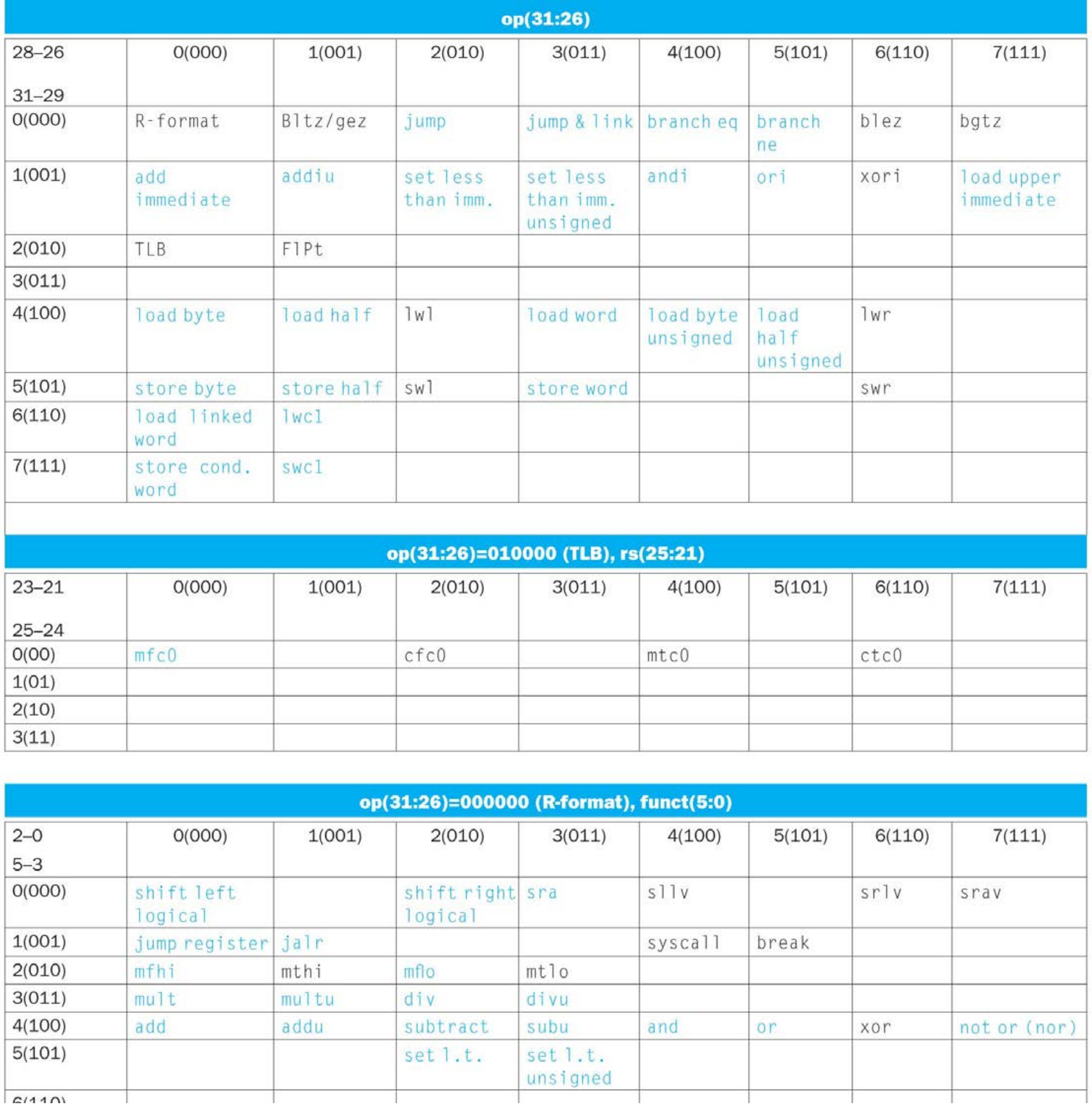
beq $s0,$s1, L1



- (Pseudo)direct addressing: jump address is 26-bit address (shift left 2 bits) concatenated with PC[31:28] (e.g. j)



Decoding MIPS instruction



op(31:26)

28–26 / 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)

23–21 / 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

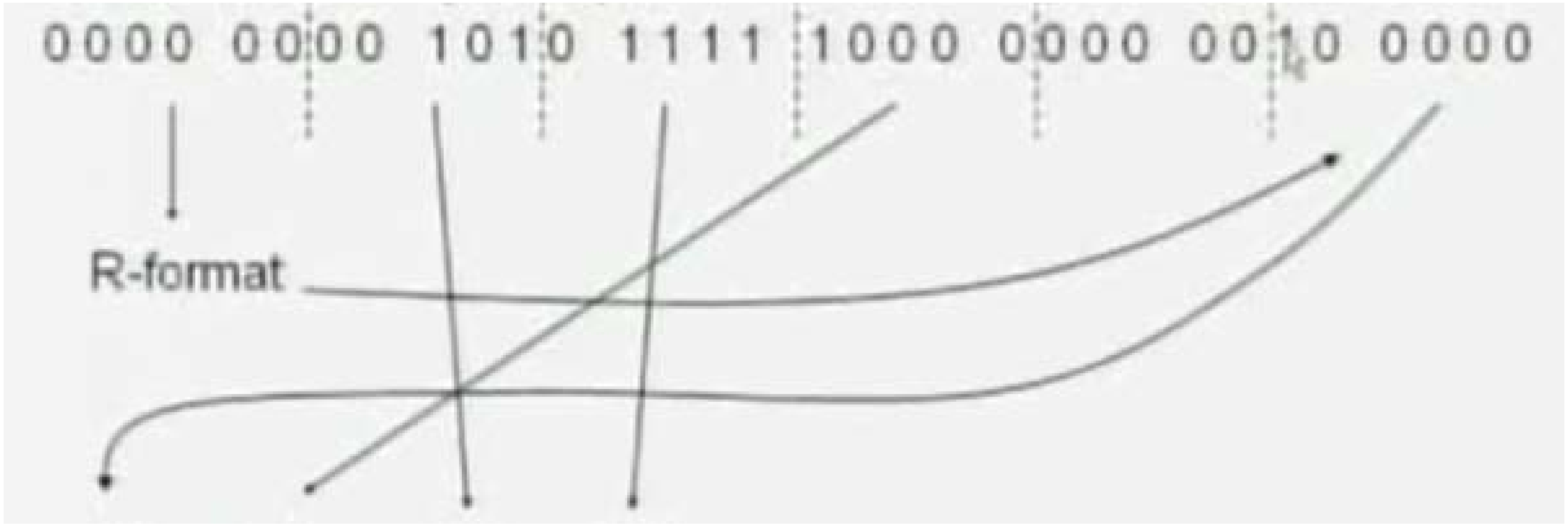
op(31:26)=000000 (R-format), funct(5:0)

2–0 / 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								

Decoding Machine Code

- What's the assembly code represent?

  00af8020 (hex)



add $s0, $a1, $t7

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- However: some useful instructions may be missing
  - Can be achieved using by other instructions
- Pseudoinstructions: figments of the assembler's imagination

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

  - $at (register 1): assembler temporary

More Pseudoinstructions in MIPS

- blt (branch less than), bgt (branch greater than), ble (branch less than and equal to), bge (branch great than and equal to)
- neg: changes the mathematical sign of the number
- not: bitwise logical negation
- li: loads an immediate value into a register

```
li $t0, 0x3BF20    ⇒    lui $t0, 0x0003
                        ori $t0, $t0, 0xBF20
```

- sge (set greater than and equal to), sgt (set great than). See Appendix A for more details

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,
  - n 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

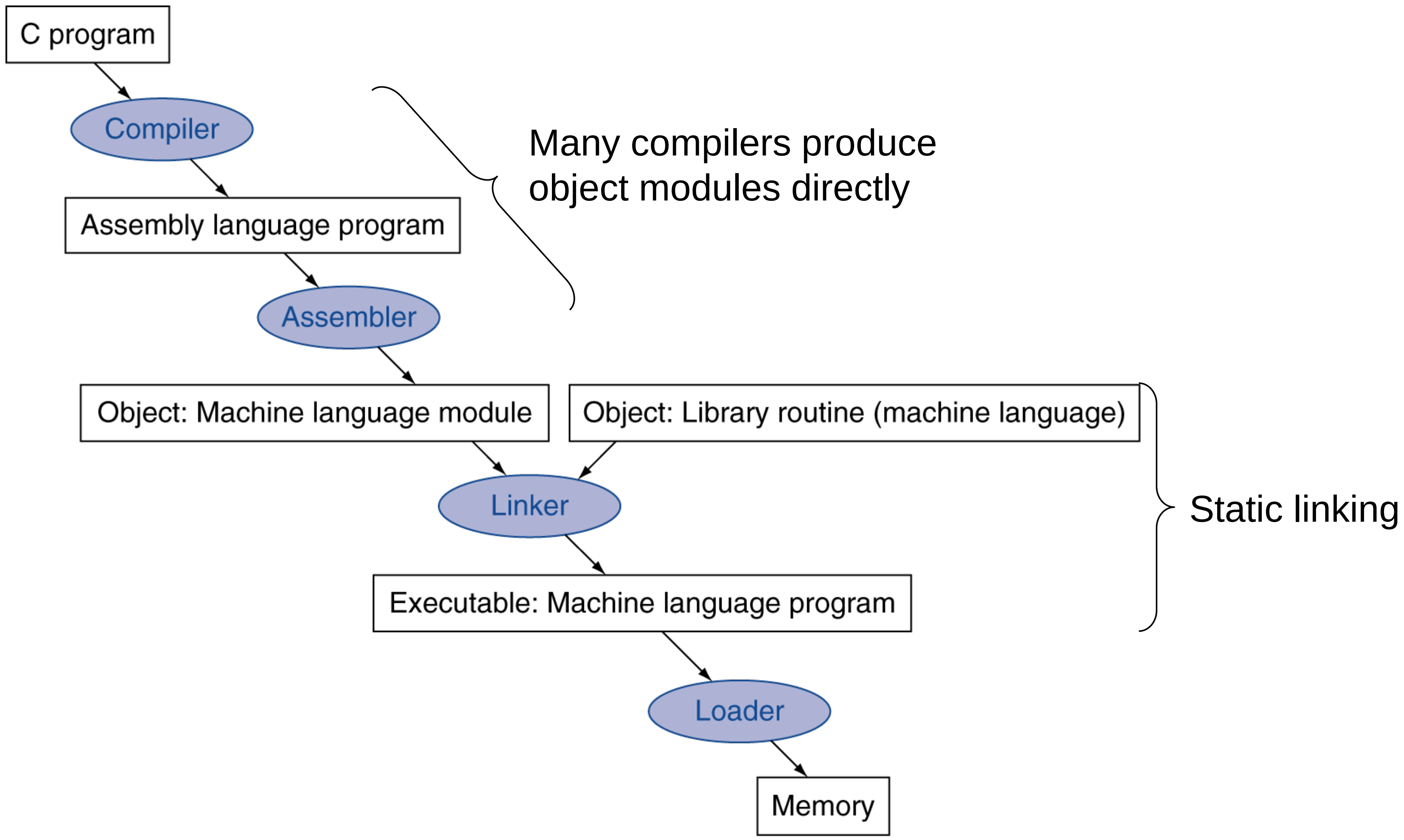
Inner loop

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
```

Program 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

Producing an Object Module

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

Linking Object Modules

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

Loading a Program

- Load from image file (executable 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() system call

Dynamic Linking

- Only link/load library procedure when it is called

ARM

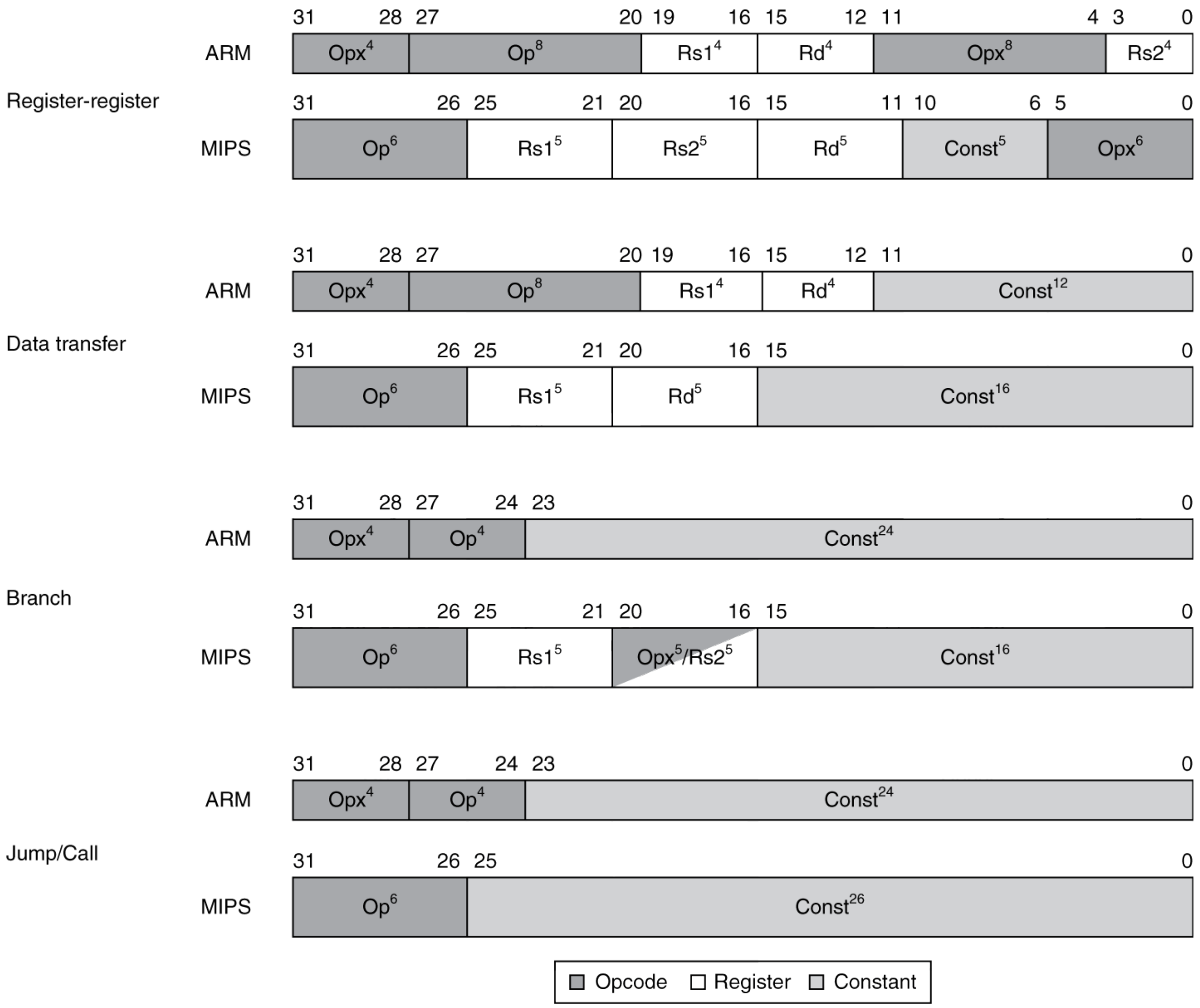
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

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

ARMv7

- Compare and Branch
  - Uses condition codes for result of an arithmetic/logical instruction
    - Negative, zero, carry, overflow (N,Z,C,V)
  - Compare instructions to set condition codes without keeping the result
  - Each instruction can be conditional
    - Top 4 bits of instruction word: condition value
- Load/Store Multiple
- Registers
  - R0-R14, R15 (PC)

Instruction Encoding (ARMv7 vs. MIPS)



ARMv8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - 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

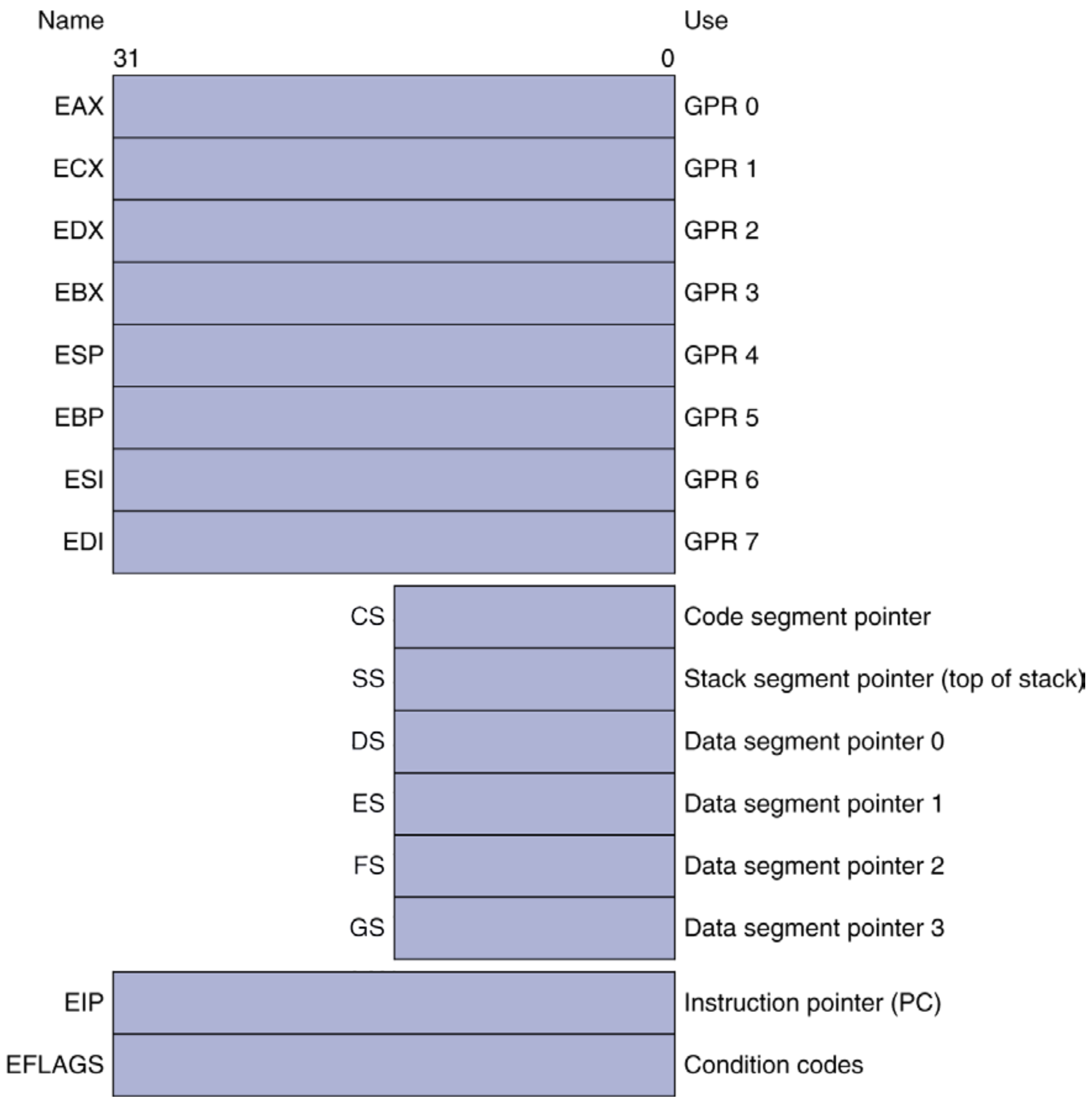
The Intel x86 ISA

- 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…
  - See the text book

Basic x86 Registers



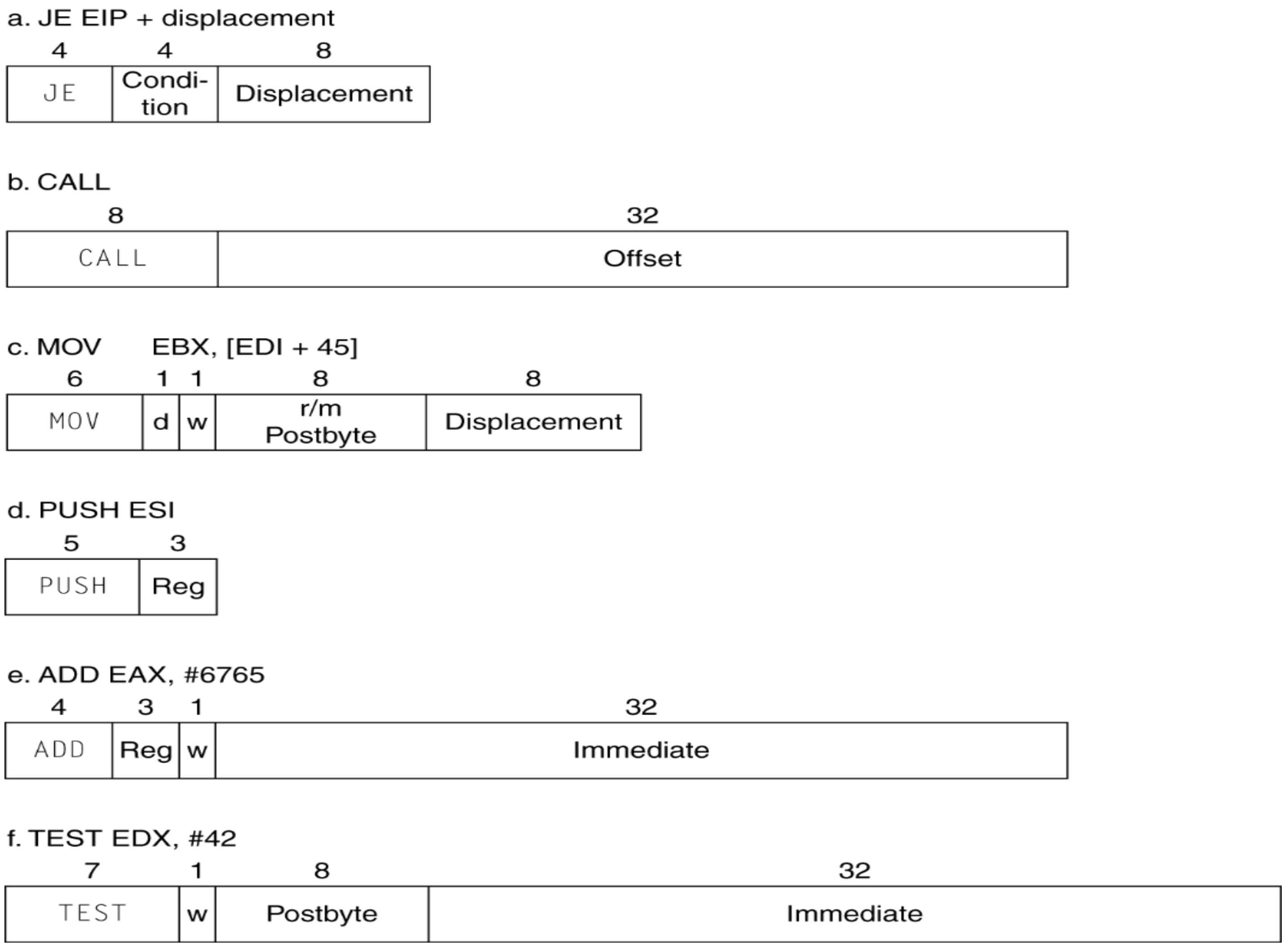
Basic x86 Addressing Modes

- 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

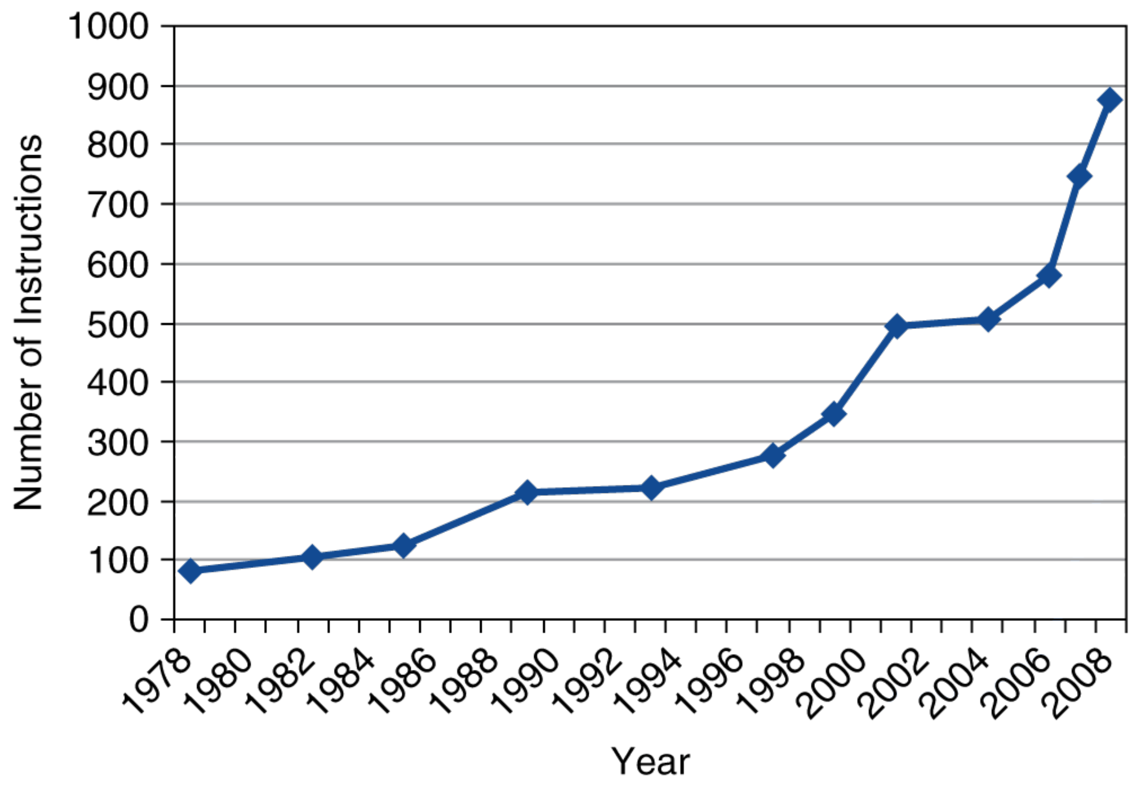


Fallacies

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

Fallacies

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



x86 instruction set

Concluding Remarks

- Design principles
  - Simplicity favors regularity
    - e.g. same instruction length
  - Smaller is faster
    - 32 registers
  - Make the common case fast
  - Good design demands good compromises
    - Same instruction length or
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs

Concluding Remarks

- Measure MIPS instruction executions 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%