Chapter 2

Instructions: Language of the Computer

Jiang Jiang
jiangjiang@ic.sjtu.edu.cn

[Adapted from *Computer Organization and Design, 4th Edition*, Patterson & Hennessy, © 2008, MK]

Instruction Set

- The collection 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

Instruction Set Architecture

- Anything programmers need to know to make a binary machine language program work correctly, including:
  - Registers
  - Organization of programmable storage
  - Data types and data structures: encoding and representations
  - Instruction set
  - Instruction formats
  - Modes of addressing: accessing data items and instructions
  - Exceptional conditions

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies
  - www.mips.com
  - Founded in 1984 by a group of researchers from Stanford University that included John L. Hennessy
- Little share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical modern ISAs
  - ARM: ARMv7
  - HP: PA-RISC 2.0
  - Intel: x86 - IA-32, x86-64 (Intel 64); IA-64
  - MIPS: MIPS32, MIPS64
  - SUN: SPARC-V9

Arithmetic Operations

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

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

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

31 … 0

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

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 operands from registers
- The size of a register in MIPS architecture is 32 bits, which is a word
- The word is the natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register

Register Operands

- The word size/length, 8b/32b/64b and the memory address space
  - GPRs(General-Purpose Registers)

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

    $s3: the size of a register
  - Cf. Width of address bus
- MIPS: 2^{32} bytes or 2^{30} words
  - Memory is byte addressed

Register Operands

- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
    - Cache of cache?
  - Numbered 0 to 31
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
  - $zero
- *Design Principle 2:* Smaller is faster
  - Cf. 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

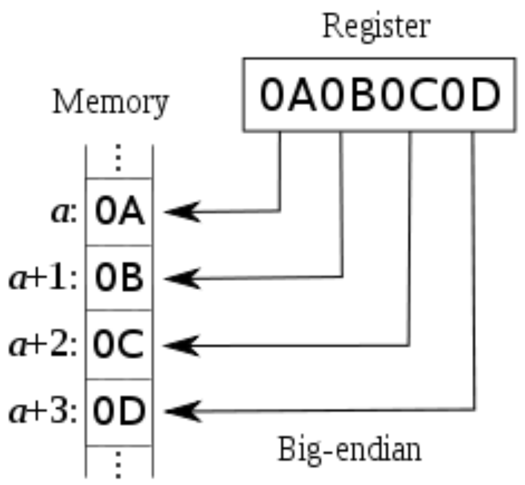
- Data transfer instruction: before arithmetic operations can be done
  - Load values from memory into registers
  - Store result from register to memory
  - RISC (Reduced instruction set computing) vs. CISC
    - Only *load* and *store* instructions access memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Alignment restriction: words are aligned in memory
  - For lw/sw, the address must be a multiple of 4
    - If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs

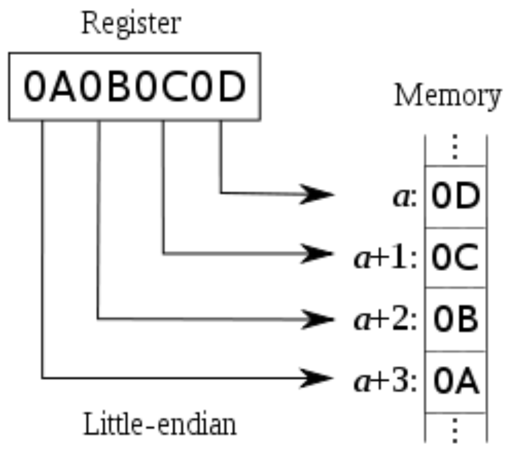
Addressing Alignment Constraints

- MIPS uses byte addressing for halfword, word, and doubleword accesses with the following alignment constraints:
  - Halfword accesses must be aligned on an even byte boundary (0, 2, 4...).
  - Word accesses must be aligned on a byte boundary divisible by four (0, 4, 8...).
  - Doubleword accesses must be aligned on a byte boundary divisible by eight (0, 8, 16...).

Endianness

- MIPS can support both big and little endian
- Big endian: most-significant byte at least address of a word.
  - Motorola 6800, 68000; PowerPC, System/370; PDP-10; SPARC until version 9
- Little endian: least-significant byte at least address
  - x86, 6502, Z80, VAX, and, largely, PDP-11
- Bi-endian: switchable endianness
  - ARM, PowerPC, Alpha, SPARC V9, MIPS, PA-RISC and IA-64





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 than memory
  - Smaller is faster
- 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 by *store*
  - 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

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

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
  - Hardwired to ground
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

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
  - 0 to +4,294,967,295

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
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 (from 0) is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- 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 -2^{n-1}
  - Most-positive: 0111 1111 … 1111 $+2^{n-1} - 1$
- $-(-2^{n-1})$ can't be represented

Signed Negation

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

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

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

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 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
- Sign Extension: replicate the sign bit to the left
  - Cf. 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

Representing Instructions

- Machine language
  - Instructions are encoded in binary, called machine code
  - Used for communication within a computer
  - Very hard to make out which instruction is which
- Assembly language
  - Symbolic representation of machine instructions： mnemonic symbol

MIPS Instruction Set

- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Similar instructions have the same format
    - Simplicity favors regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

MIPS R-format Instructions

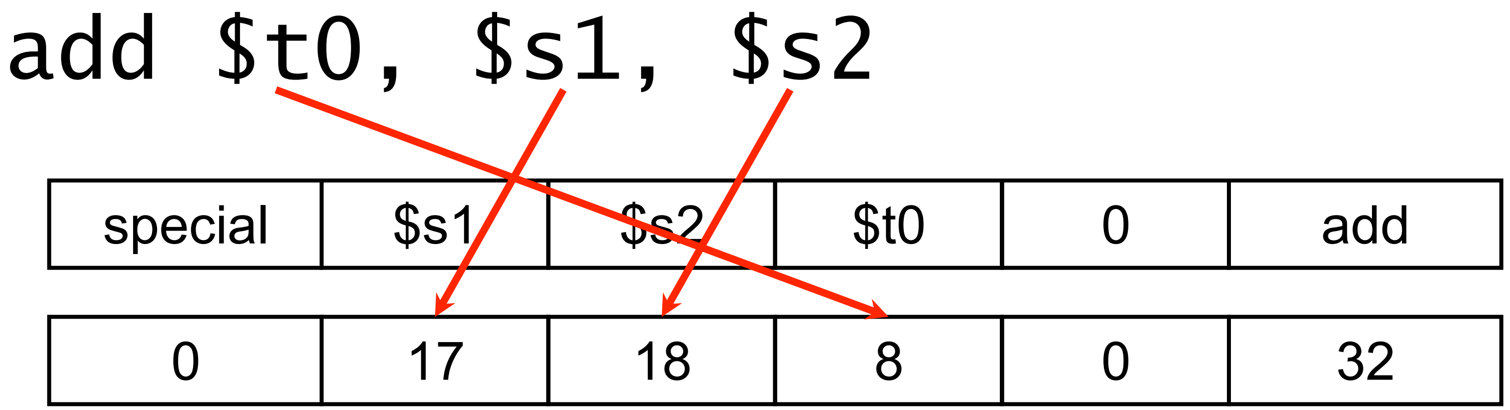
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction format
  - The layout of the instruction
- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000, useless for R-format)
  - 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

$00000010001100100100000000100000_2 = 02324020_{16}$

Machine code

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

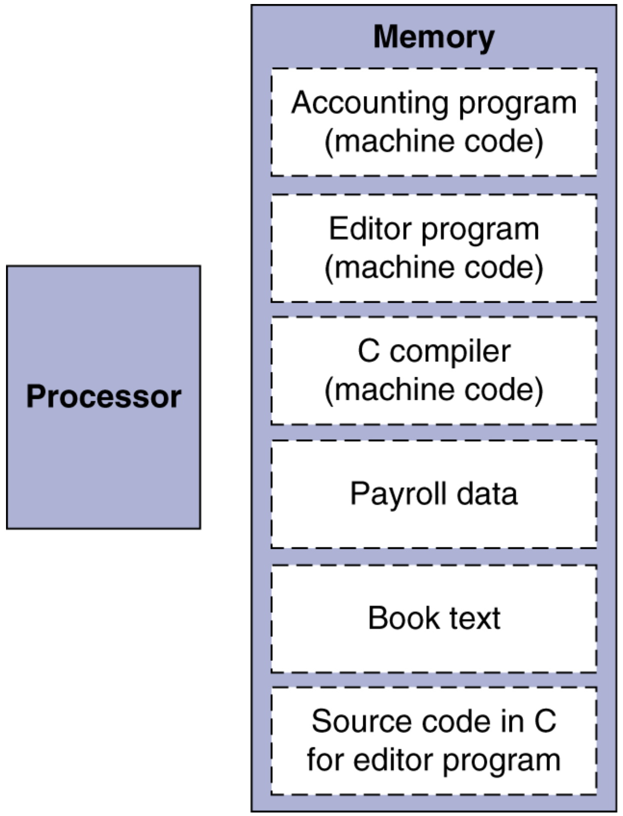
MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rt: destination (addi, lw) or source (sw) register number
  - Constant: -2^{15} to $+2^{15} - 1$ (signed)
  - 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

Stored Program Computers

The BIG Picture

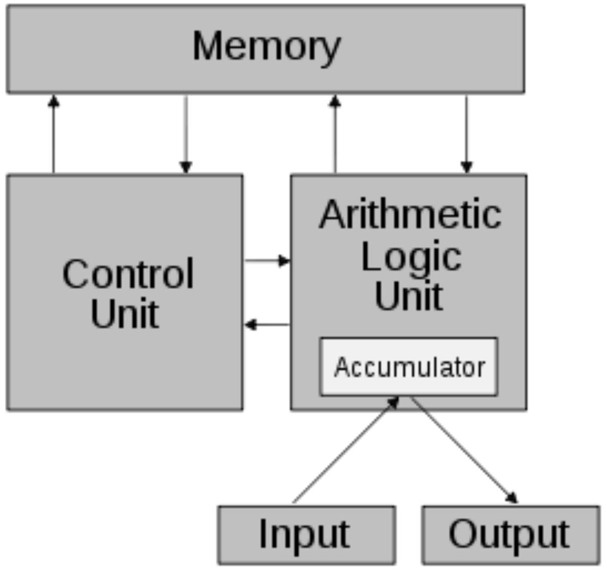


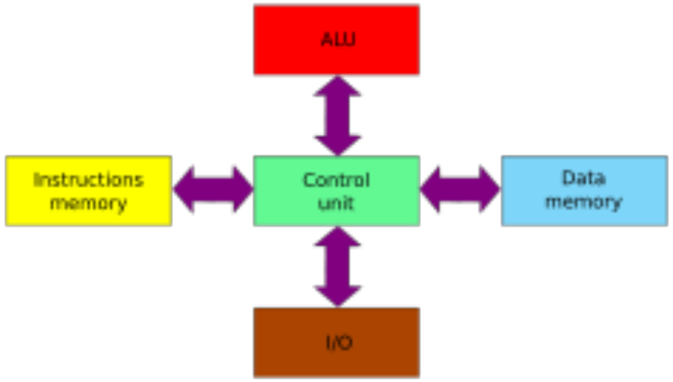
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Stored-program concept
  - Instructions represented in binary as numbers, just like data
  - Instructions and data are stored in memory to be read or written, just like numbers
- Programs can operate on programs
  - E.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Often leads industry to align around a small number of ISAs, e.g. x86, ARM
  - Ecosystem

Von Neumann architecture

- Von Neumann architecture
  - “Stored-program computer“ and "Von Neumann architecture" are interchangeably
  - Named after the mathematician and early computer scientist John von Neumann
  - Arose from Von Neumann's paper "First Draft of a Report on the EDVAC"
- Von Neumann bottleneck
  - The limited throughput between the CPU and memory (memory wall)
- Cf. Harvard architecture



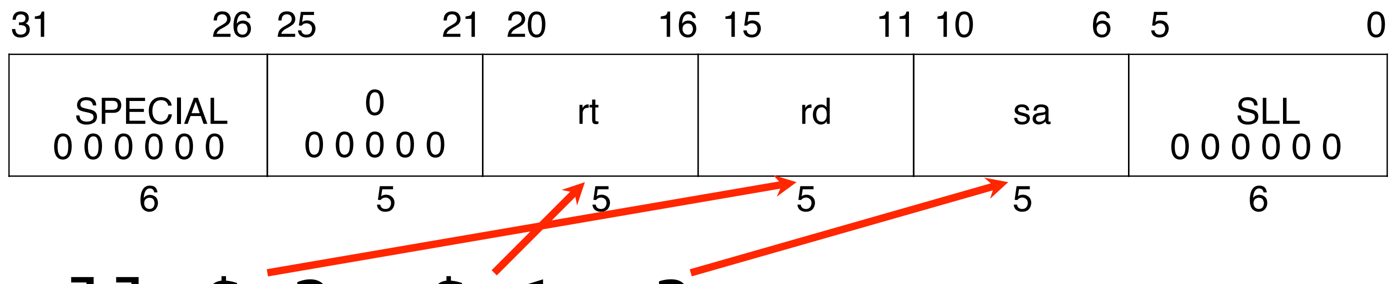


Logical Operations

- 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`

Shift Operations



```
sll $t2, $t1, 3
```

- R-format
  - rs: unused
  - shamt: how many positions to shift
    - Useless for add/sub
  - Simplicity favors regularity!
- Useful for extracting and inserting groups of bits in a word

Shift Operations

- 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)
- Shift right arithmetic
  - Duplicate the sign-bit (bit 31)
  - `sra` by *i* bits divides by 2^i (signed)

AND Operations

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

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	AND 1 0 0 1 0 0
6	5	5	5	5	6

```
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

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	OR 1 0 0 1 0 1
6	5	5	5	5	6

```
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
  - No NOT operation, but NOR
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	NOR 1 0 0 1 1 1
6	5	5	5	5	6

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

$t1 0000 0000 0000 0000 0011 1100 0000 0000

$t0 1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

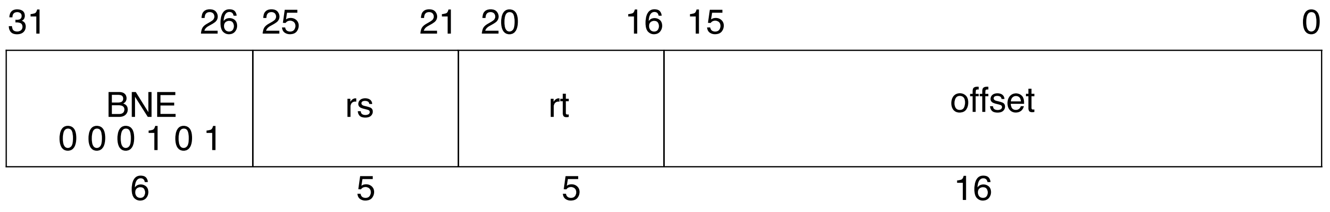
- The difference between a computer and a calculator is the ability to make decisions
- In high level language
  - *if* statement
  - *go to* statement
- Branch to a labeled instruction if a condition is true; Otherwise, continue sequentially
- Conditional and unconditional branches
- PC-relative and absolute/register indirect branch

Conditional Operations

- MIPS defines the following instructions
  - PC-relative conditional branch: bne,beq, ± 128 KB
  - PC-region unconditional jump: j, jal, 256MB region
  - Absolute/register indirect unconditional jump: jr
- How to “decide”? Program counter (PC)
  - The register containing the address of the instruction in the program being executed
  - PC is affected only indirectly by certain instructions - it is NOT an architecturally-visible register

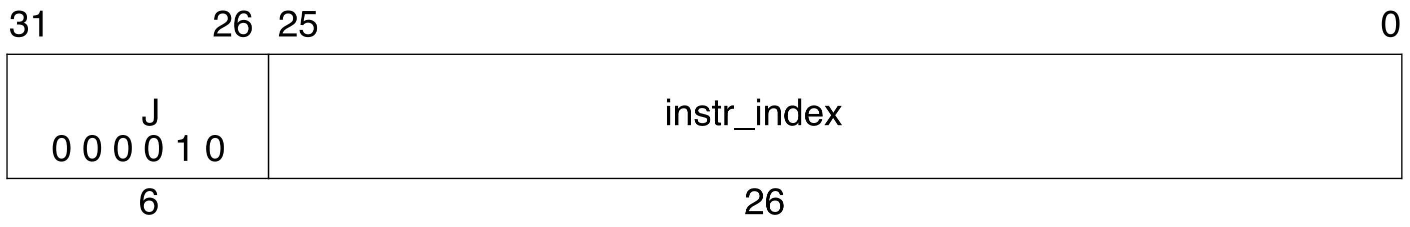
Conditional Operations

- 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;
  - Cf. ARM
    - `cmp r1, r2      #CPSR <= r1-e2 in`
    - `beq L1          #test CPSR, then..`



Conditional Operations

- Unconditional branch
  - `j L1`
    - Unconditional jump to instruction labeled L1
    - PC-region: 256 MB aligned region

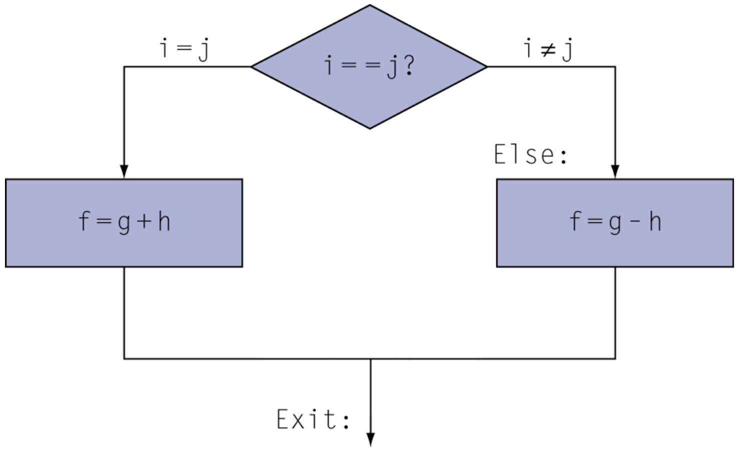


Compiling If Statements

- 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:

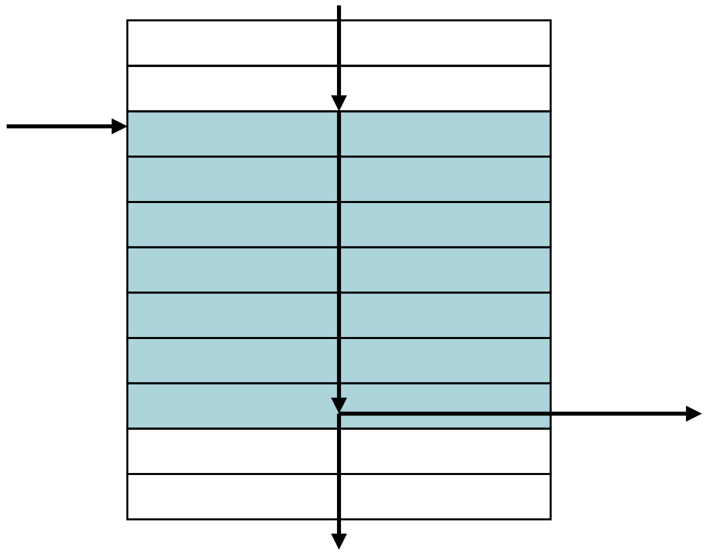
```
while (save[i] == k) i += 1;      //cf. for
```

  - i in $s3, k in $s5, base address saved in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2      #word:4B,*4
      add  $t1, $t1, $s6    #base+offset*4
      lw   $t0, 0($t1)      #load save[i]
      bne  $t0, $s5, Exit   # <> k ?
      addi $s3, $s3, 1      # ==k, i += 1
      j    Loop
Exit: …
```

Basic Blocks

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



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

More Conditional Operations

- Set on less than: `slt`
  - Set result to 1 if a condition is true
  - Otherwise, set to 0

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SLT 1 0 1 0 1 0
6	5	5	5	5	6

- `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 ➔ `blt`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  # branch to L, ≠ 0
```

Branch Instruction Design

- Why not `blt`, `bge`, etc? (just `beq`, `bne`)
  - Pseudoinstruction
- 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
  - Make the common case fast
- This is a good design compromise
  - Good design demands good compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltiu`
- 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

Pseudoinstruction

Name	instruction syntax	Real instruction translation	meaning
Move	move $rt,$rs	addi $rt,$rs,0	R[rt]=R[rs]
Clear	clear $rt	add $rt,$zero,$zero	R[rt]=0
Not	not $rt, $rs	nor $rt, $rs, $zero	R[rt]=~R[rs]
Load Address	la $rd, LabelAddr	lui $rd, LabelAddr[31:16]; ori $rd,$rd, LabelAddr[15:0]	$rd = Label Address
Load Immediate	li $rd, IMMED[31:0]	lui $rd, IMMED[31:16]; ori $rd,$rd, IMMED[15:0]	$rd = 32 bit Immediate value
Branch unconditionally	b Label	beq $zero,$zero,Label	if(R[rs]==R[rt]) PC=Label
Branch and link	bal $rs,Label	bgezal $zero,Label	if(R[rs]>=0) PC=Label
Branch if greater than	bgt $rs,$rt,Label	slt $at,$rt,$rs; bne $at,$zero,Label	if(R[rs]>R[rt]) PC=Label
Branch if less than	blt $rs,$rt,Label	slt $at,$rs,$rt; bne $at,$zero,Label	if(R[rs]<R[rt]) PC=Label
Branch if greater than or equal	bge $rs,$rt,Label	slt $at,$rs,$rt; beq $at,$zero,Label	if(R[rs]>=R[rt]) PC=Label
Branch if less than or equal	ble $rs,$rt,Label	slt $at,$rt,$rs; beq $at,$zero,Label	if(R[rs]<=R[rt]) PC=Label
Branch if greater than unsigned	bgtu $rs,$rt,Label		if(R[rs]>R[rt]) PC=Label
Branch if greater than zero	bgtz $rs,Label		if(R[rs]>0) PC=Label
Branch if equal to zero	beqz $rs,Label		if(R[rs]==0) PC=Label
Multiplies and returns only first 32 bits	mul $d, $s, $t	mult $s, $t; mflo $d	$d = $s * $t
Divides and returns quotient	div $d, $s, $t	div $s, $t; mflo $d	$d = $s / $t
Divides and returns remainder	rem $d, $s, $t	div $s, $t; mfhi $d	$d = $s % $t

Procedure

- Procedure
  - A stored subroutine that performs a specific task based on the parameters
  - Advantages: code reuse, easy understanding
- Caller: the calling procedure, providing parameters to callee
- Callee: the called program, executes stored instructions based on the parameters provided by the caller

Procedure Calling

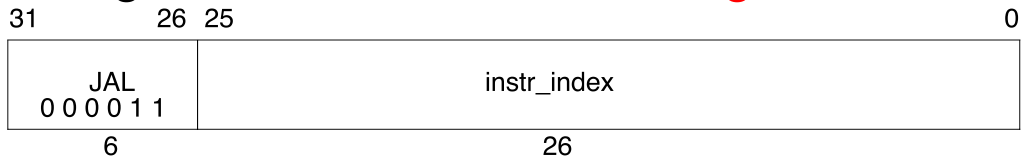
- To execute a procedure, steps required
  1. Place parameters in registers accessible to procedure (callee)
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register accessible to caller
  6. Return to place of call

Register Usage

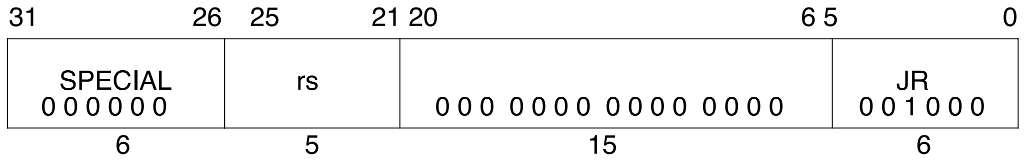
- $a0 – $a3: reg's 4 – 7
  - 4 arguments register to pass parameters
- $v0, $v1: reg's 2 – 3
  - 2 result registers to return values
- $ra: reg 31
  - return address register to return to the point of call
- $t0 – $t9: reg's 8 – 15, reg's 24 – 25
  - Temporaries. Can be overwritten by callee
- $s0 – $s7: reg's 16 – 23
  - 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)
- $at: assembler temporary (reg 1)

Procedure Call Instructions

- Procedure call: jump and link
  `jal ProcedureLabel`
  - Address of following instruction put in $ra
    - Link: put the return address in $ra ($31)
    - The same procedure can be called from several parts (address) of the program
  - Jumps to target address: in 256 MB region



- Procedure return: jump register
  `jr $ra`
  - Copies $ra to program counter: register indirect branch, 32b
  - Unconditional branch to the address of $ra

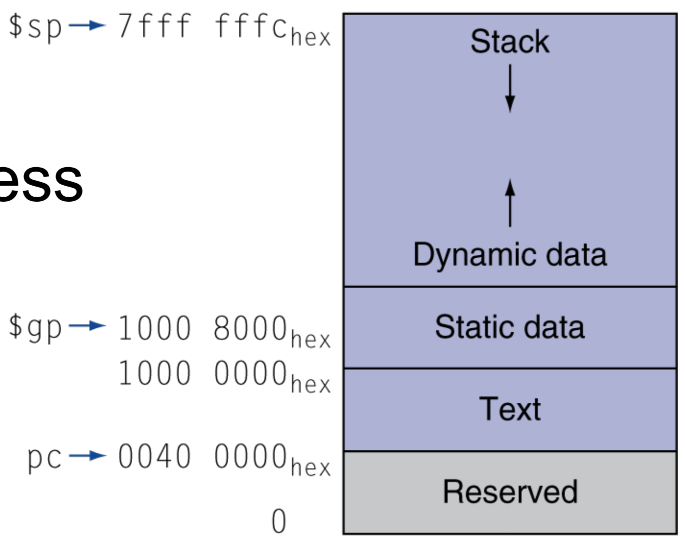


Procedure Call

- Caller
  - The calling program puts the parameter values in \$a0 - \$a3 (4 registers)
  - Transfers control to jump to procedure x using `jal x`
- Callee
  - The procedure x then performs the calculations
  - Places the results in \$v0 and \$v1 (2 registers)
  - Returns control to caller using `jr $ra`

Using More Registers

- What if more register for a procedure needed?
  - To spill register to memory from register file
  - Ideal data structure is a stack
- Stack: a last-in-first-out queue in memory
  - Stack pointer ($sp, reg 29) to most recently allocated address
  - Stacks "grow" from higher to lower address
  - Operations
    - push: by subtracting from $sp
    - pop: by adding to $sp



Temporary and Saved Register

- MIPS software separates 18 registers into
  - $t0 – $t9: temporaries registers
    - Can be overwritten (not preserved) by callee
    - The caller will never use the values in them
  - $s0 – $s7: saved registers
    - Must be saved/restored by callee if callee wants to use them
    - The caller need to use the values in them again
- This convention(!) reduces register spilling

Leaf Procedure Example

- Procedures that do not call others
- 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
   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: spill

Procedure body

Move result to $v0

Restore $s0

Return

Non-Leaf Procedures

- Procedures that call other procedures
  - Parent and child at the same time
- For nested call
  - The caller (may be called) needs to save on the stack
    - Any arguments ($a0-$a3) : from parent, may be used after the call
    - Any temporaries ($t0-$t9) needed after the call: may be used by child freely
  - The callee (may call) needs to push to the stack
    - Its return address ($ra) : may be used by child to call his child
    - Any saved registers ($s0-$s7) used by the callee: from parent
- On the return, the registers are restored from memory and the $sp is readjusted

Non-Leaf Procedure Example

- C code: calculate factorial

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

  - Argument n in $a0
  - Result in $v0

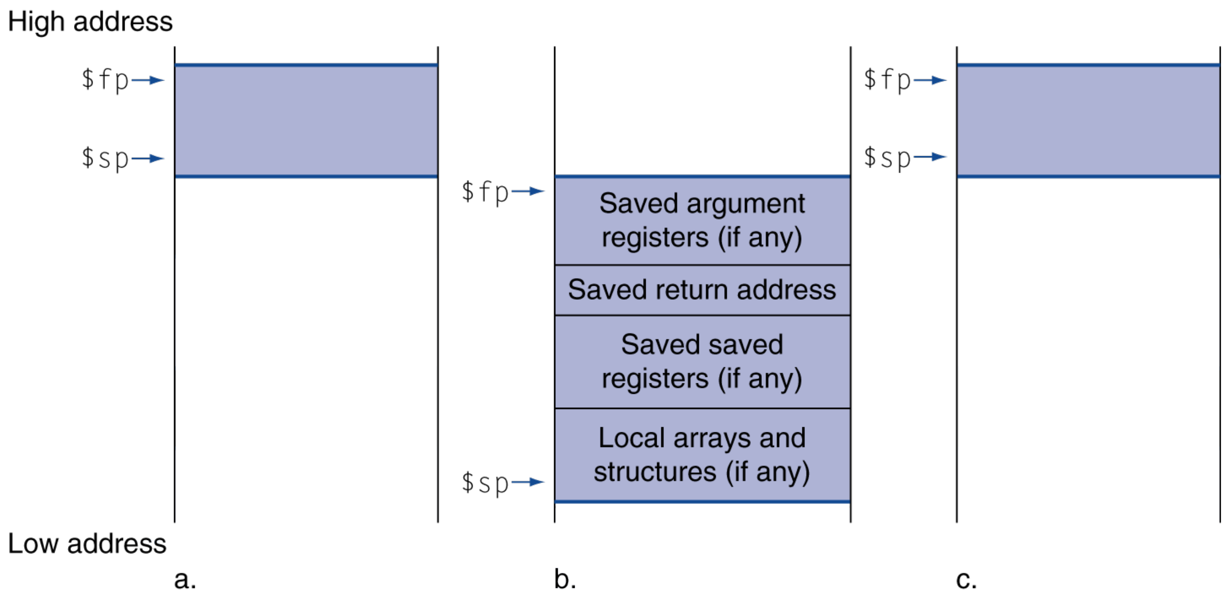
Non-Leaf Procedure Example

- MIPS code: act as both caller and callee

```
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
```

Use $a0 after the call

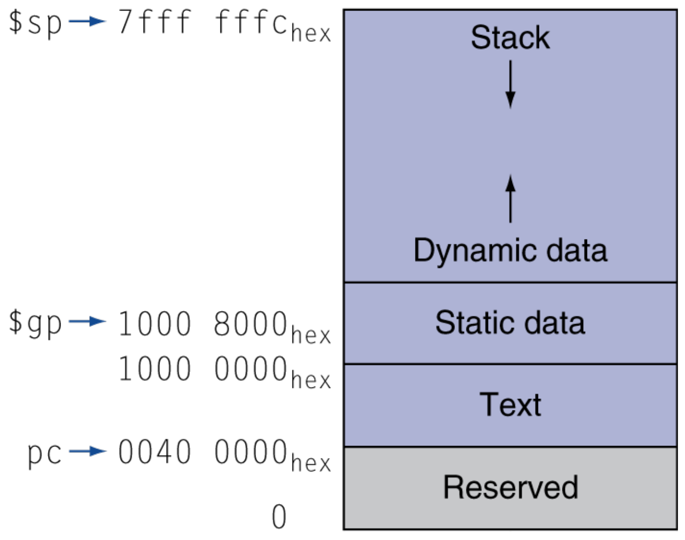
Local Data on the Stack


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.

- Procedure frame (activation record)
  - The segment of stack containing a callee's saved registers and local variables (allocated by callee) in memory, e.g., C automatic variables
  - Used by some compilers to manage stack storage
  - Frame pointer ($fp, reg 30): point to first word of the frame of a procedure

Memory Layout

- Text segment
  - Machine code
- Static data segment
  - Global variables, e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets
- Dynamic data segment
  - Heap: grow from low to high
  - E.g., malloc()/free() in C, new in java
- Stack: automatic storage



Heap and stack grow toward each other, allowing more efficient use of memory

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - American Standard Code for Information Interchange
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 16/32-bit character set
  - Universal encoding of the alphabets of most human language
  - Used in Java (16 bit by default), C++ wide characters
  - Most of the world's alphabets, plus symbols

Byte/Halfword Operations

- Could use bitwise operations
  - Byte: `lb`, `lbu`, `sb`;
  - Halfword: `lh`, `lhu`, `sh`
- 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

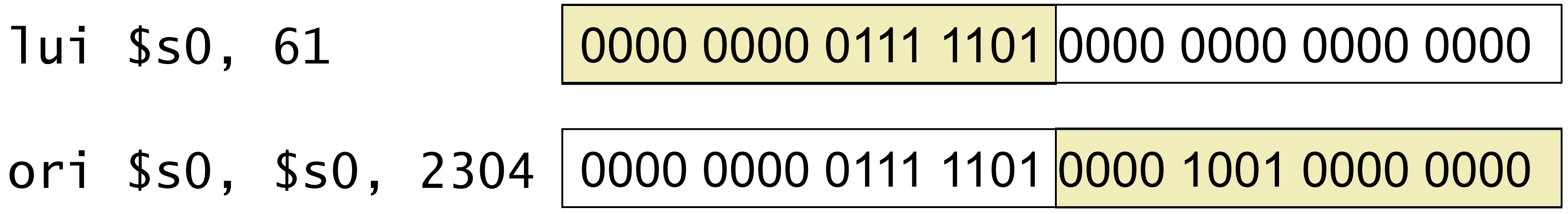
String Copy Example

- 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 (common case)
  - `lui rt, constant (16b)`
    - Load Upper Immediate
    - Copies 16-bit constant to left 16 bits of rt
    - Clears right 16 bits of rt to 0
  - `ori rt, rs, constant (16b)`
    - or immediate
- How to get a 32b constant?
  - 0000 0000 0011 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

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Target address = PC + offset × 4
  - Forward or backward: signed
  - With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes.

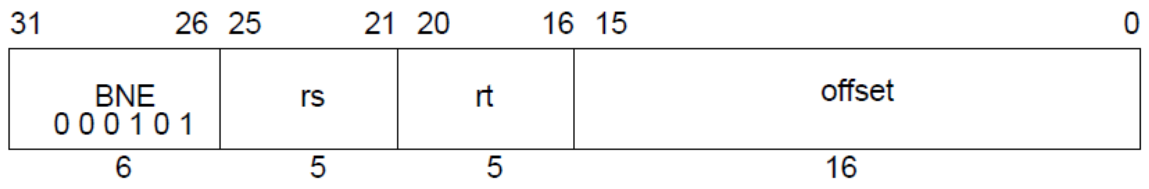
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Branch Addressing

- PC-relative addressing
  - Target address = PC + offset × 4
- PC already incremented by 4 by this time (Chapt. 4)
  - It’s convenient for HW to increment the PC early to point to the next instruction
  - The offset is relative to the next instruction (pc + 4), not the current instruction (pc) – the distance, the number of instructions
  - For branch instructions, the address of the instruction in the delay slot (not the branch itself)

Branch Implementation

- BNE
  - Format: `bne rs, rt, offset`



  - Operation

```
I:      tgt_offset ← sign_extend(offset || $0^2$)
        condition ← (GPR[rs] ≠ GPR[rt])
I+1 :if condition then
            PC ← PC + tgt_offset
        endif
```

Instruction word address

  - The offset is signed: the 18-bit offset, ± 128 KB range
- PC-relative conditional branch
  - PC = PC + offset (sign_extend)

Jump Addressing

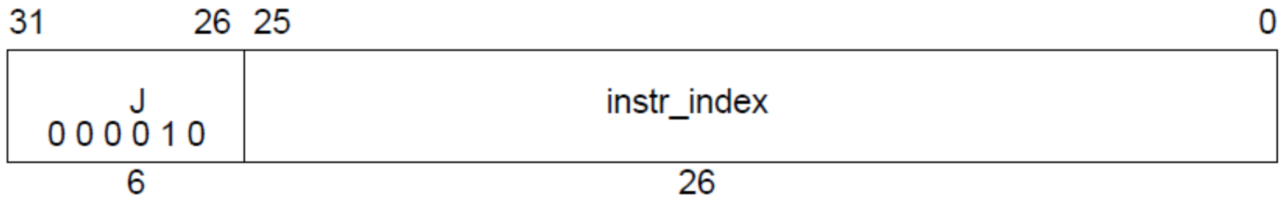
- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- PC-region unconditional branch
  - Target address = $PC_{31\ldots28}$: (address × 4)
  - With a 28 bits offset, to branch within the current 256 MB aligned region
    - A 256 MB region aligned on a 256 MB boundary

Branch Implementation

- J
  - Format: J target



  - Operation

$$\text{I:}$$
$$\text{I+1:} PC \leftarrow PC_{GPRLEN..28} \,||\, instr\_index \,||\, 0^2$$

  - The offset is unsigned

Instruction word address

Target Addressing Example

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

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

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

Branching Far Away

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

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

↓ replaced by

```
        bne $s0, $s1, L2 #L2, (16+2)b
        j L1             #L1, (26+2)b
    L2: …
```

Jump and Branch Instructions

- MIPS defines the following jump and branch instructions:
  - PC-relative conditional branch: bne,beq
  - PC-region unconditional jump: j, jal
  - Absolute (register) unconditional jump: jr
  - A set of procedure calls that record a return link address in a general register: jal, jr

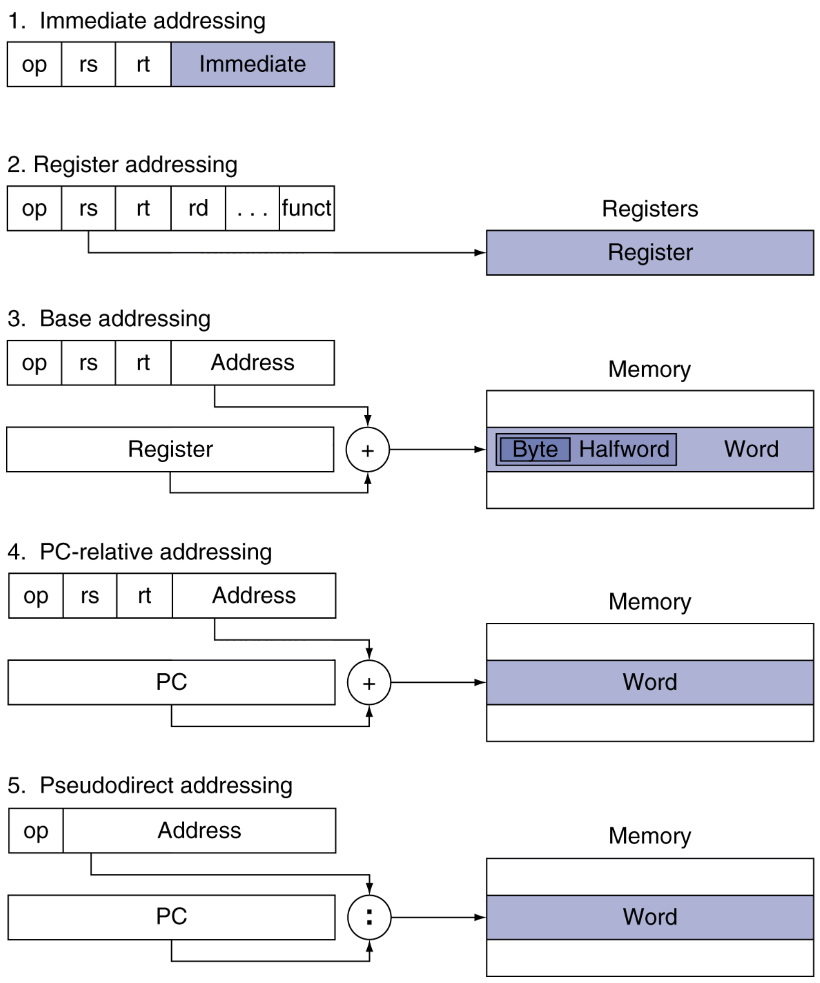
Jump and Branch Instructions

Mnemonic	Instruction	Location to Which Jump Is Made	Defined in MIPS ISA
J	Jump	256 Megabyte Region	MIPS32
JAL	Jump and Link	256 Megabyte Region	MIPS32
JALR	Jump and Link Register	Absolute Address	MIPS32
JALR.HB	Jump and Link Register with Hazard Barrier	Absolute Address	MIPS32 Release 2
JALX	Jump and Link Exchange	Absolute Address	MIPS16e
JR	Jump Register	Absolute Address	MIPS32
JR.HB	Jump Register with Hazard Barrier	Absolute Address	MIPS32 Release 2

Addressing Mode

- How the instructions identify the operand (or operands) of each instruction
- An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.

Addressing Mode Summary



1. ADDI rt, rs, immediate
   `addi $s1, $s2, 3`

2. ADD rd, rs, rt
   `add $s1, $s2, $s3`

3. LW rt, offset(base)
   `lw $s1, 20($s2)`

4. BNE rs, rt, offset
   `bne $s1, $s2, 25`

5. J target
   `j 2500`

MIPS Addressing: Register operand, Immediate operand, Register + offset

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 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, e.g. `ll`, `sc`

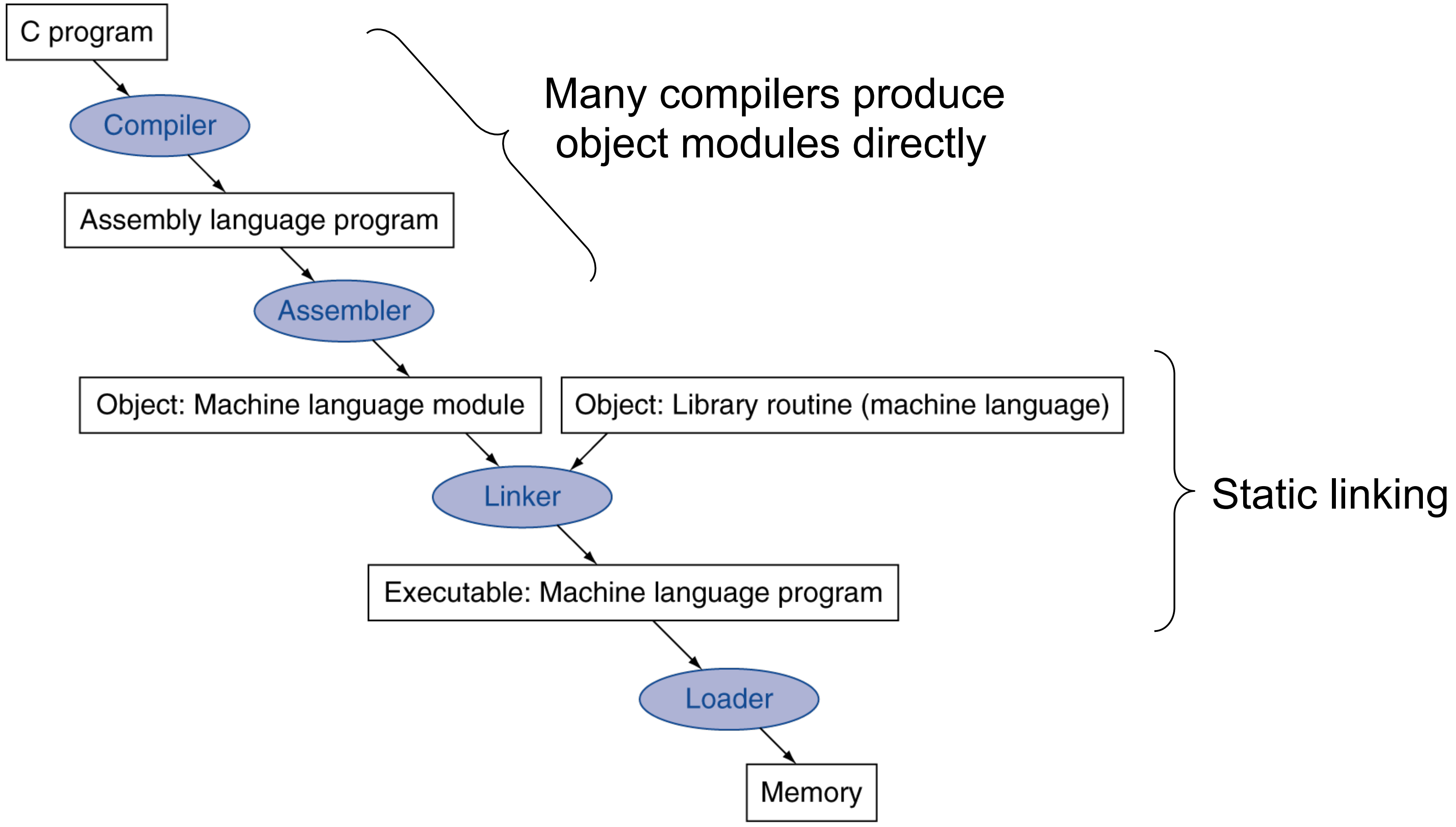
Synchronization in MIPS

- 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 ;fails, try again
     add $s4,$zero,$t1 ;put load value in $s4
```

Translation and Startup

Four steps:



C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Assembler Pseudoinstructions

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

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

  - $at (register 1): assembler temporary

ARM & MIPS Similarities

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

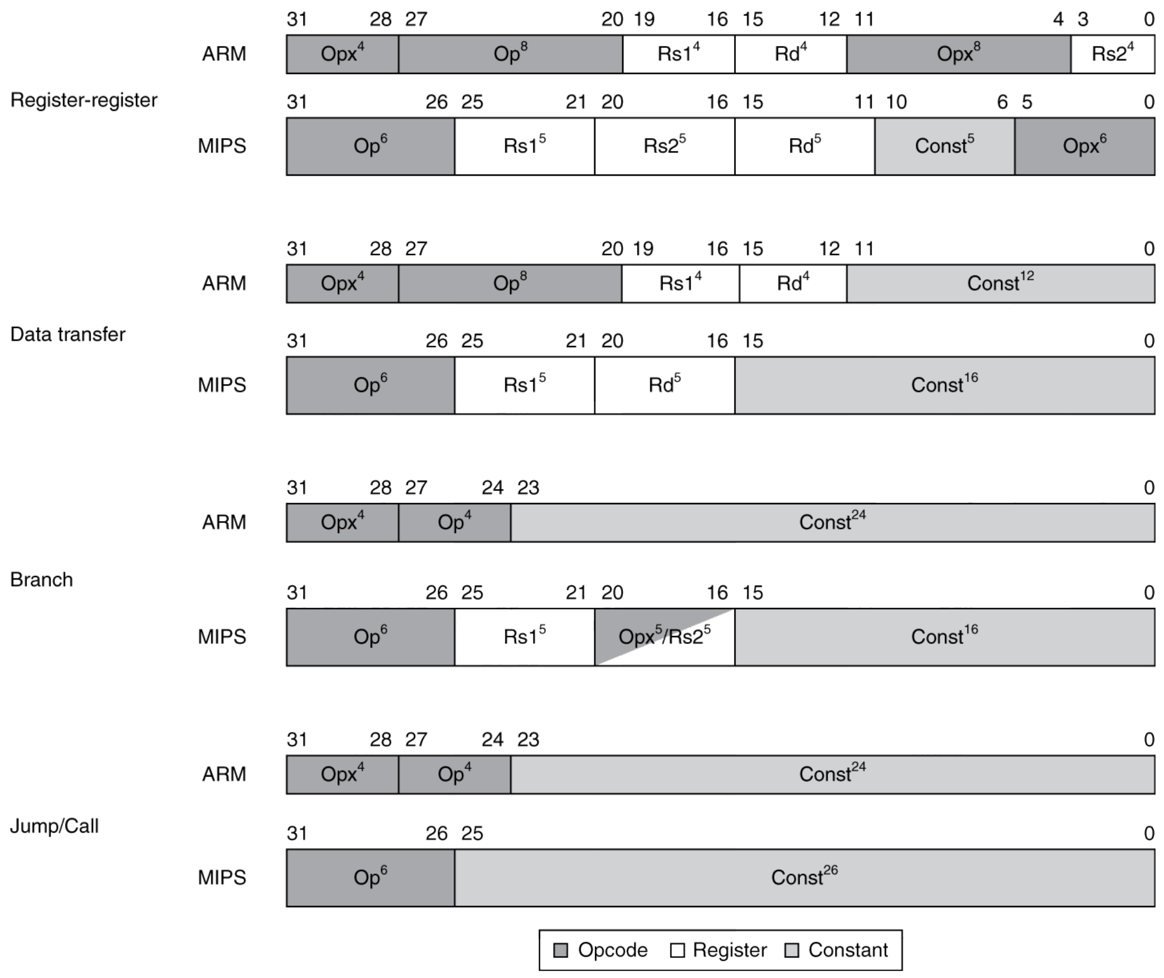
	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

MIPS Addressing: Register operand, Immediate operand, Register + offset

Compare and Branch in ARM

- ARM uses condition codes for result of an arithmetic/logical instruction
  - 4 bits in program status word:
    - 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
  - Cf. Predict, PRF in IA-64

Instruction Encoding



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

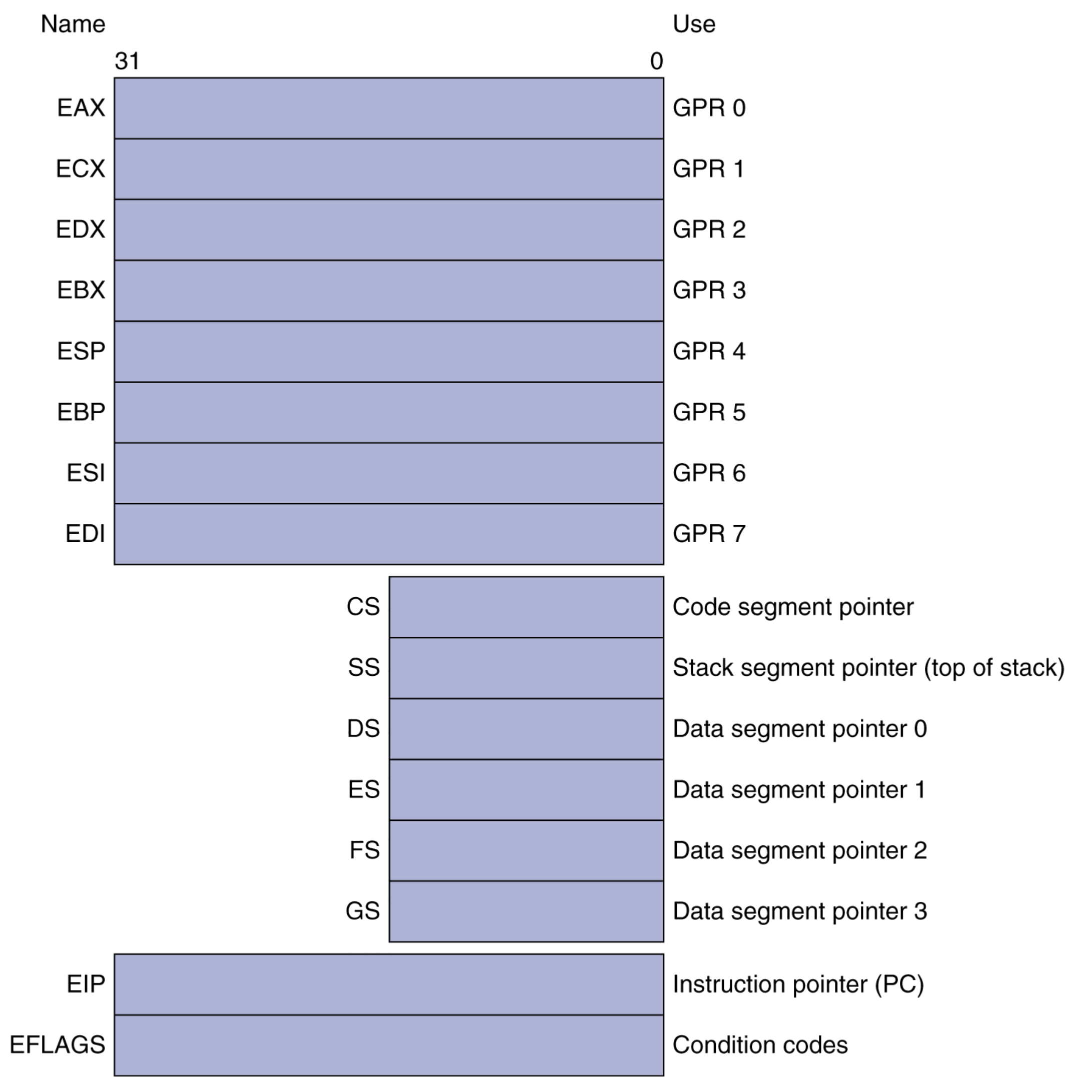
The Intel x86 ISA

- 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



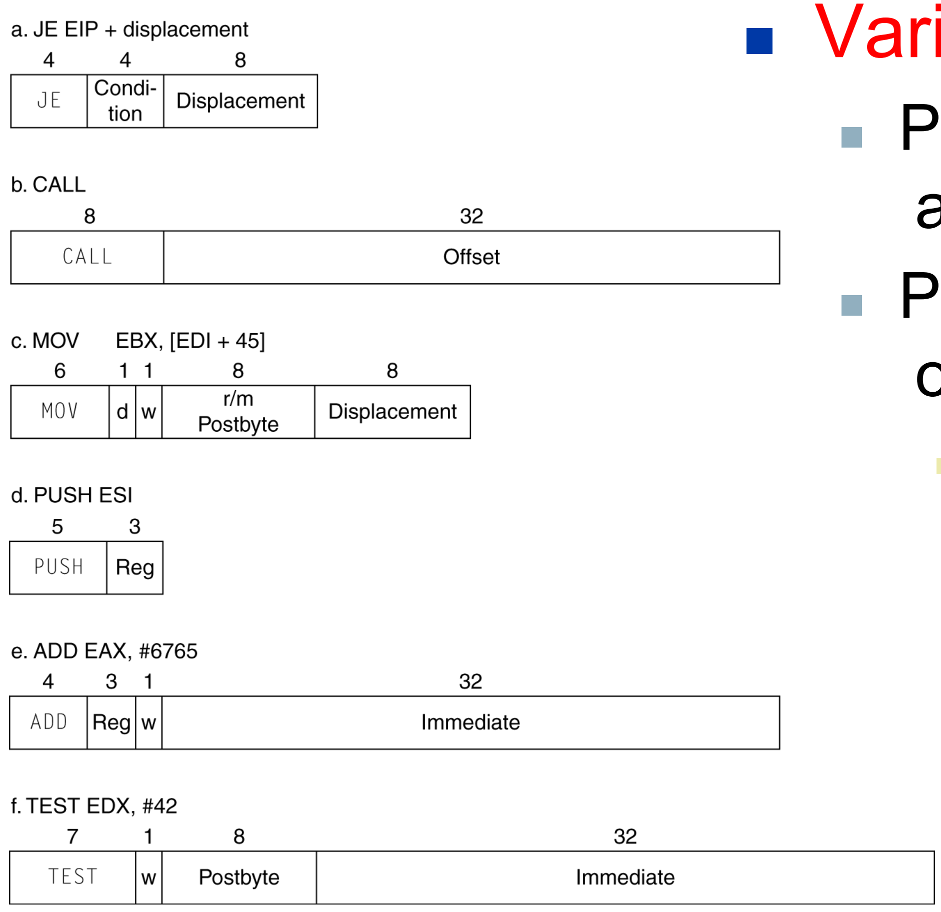
Basic x86 Addressing Modes

- 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

Fallacies

- Fallacy: More powerful instructions means 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

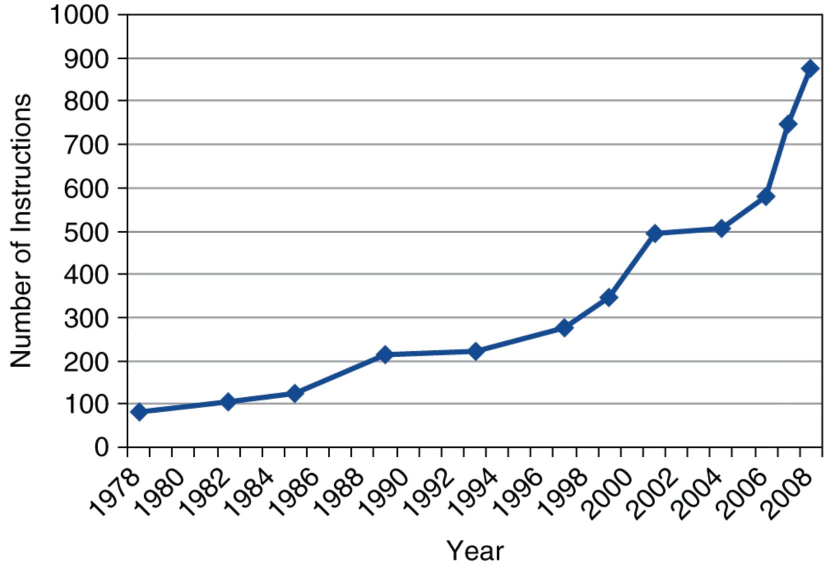
Fallacies

- Fallacy: Write in assembly language to obtain the highest performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity

Fallacies

- Fallacy: The importance of commercial binary compatibility means successful instruction set don’t change
  - Backward compatibility is sacrosanct !
  - But they do accrete more instructions
  - X86: 1 instruction/month !

x86 instruction set



Pitfalls

- Pitfall: Forgetting that sequential word address in machines with byte addressing do not differ by one
  - Sequential words addresses are not at sequential byte addresses
  - Increment by 4, not by 1!

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
  - Cf. 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%