



## Topic 2

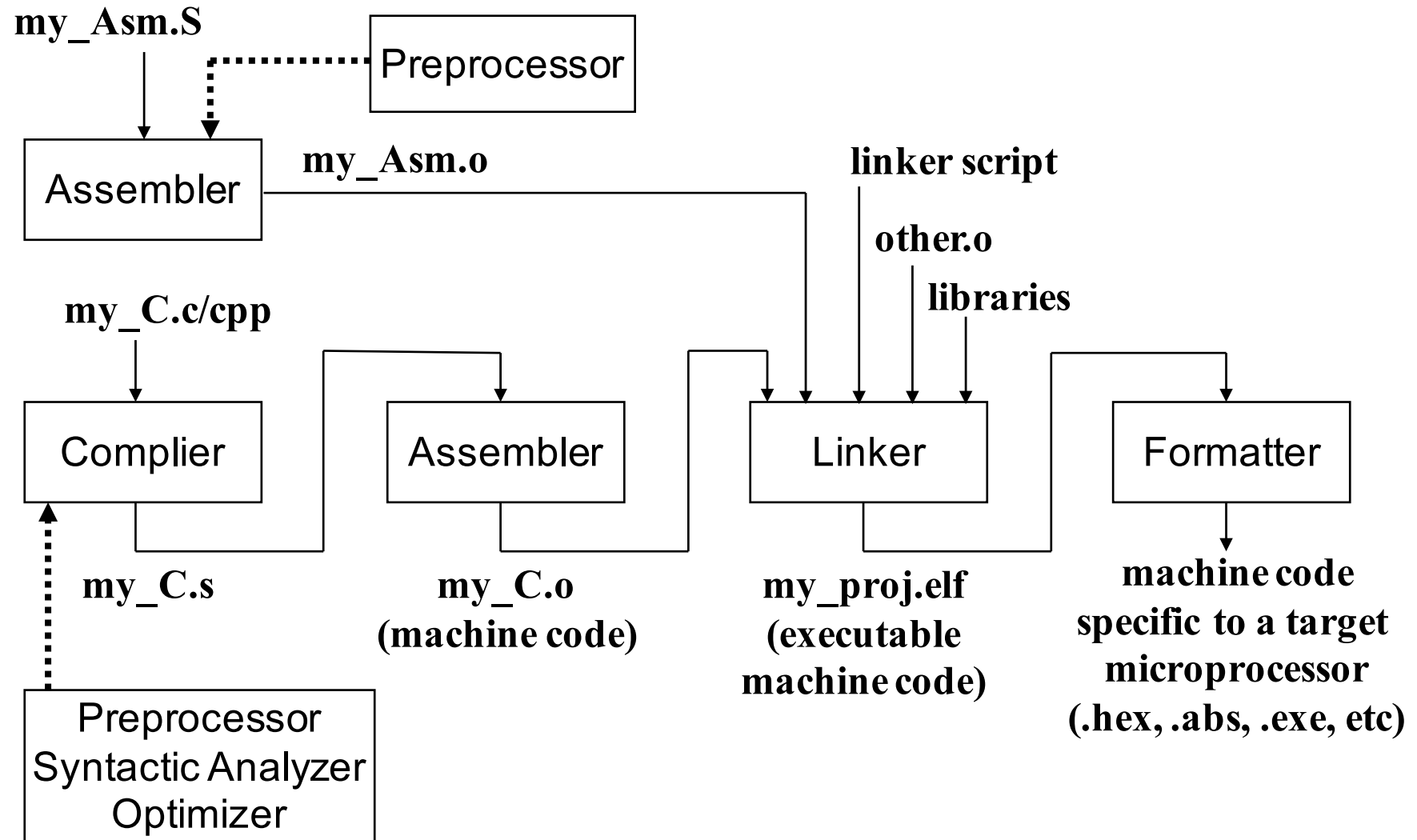
### **Assembly Programming - Operations and Operands**

# Levels of Program Code

- High-level language (translator: compiler)
  - Level of abstraction closer to problem domain
  - Provides productivity and portability
- Assembly language (translator: assembler)
  - Low-level language
  - Symbolic representation of binary machine code
  - Direct correspondence to machine code
  - More readable than machine code
- Machine language
  - Binary digits (bits) – language of digital circuits
  - Composed of instructions (commands for computer) and data
  - Instructions and data encoded in binary digitals



# Processing Different Languages



# Assembly Language

- When to use?
  - Compilers introduce uncertainty about execution time and size
  - Use when speed and size of program are critical
  - Can mix high-level language with assembly
- Drawbacks of Assembly language
  - Can be very time consuming
  - No assembler optimization
  - Almost impossible to be portable
    - Different computers support different assembly languages that requires different assembler
    - Assembly languages are similar
  - Hard to debug



# Instruction Set

- or ISA, all commands that computer understands including data types, registers, addressing modes, etc.
- Different computers have different instruction sets
  - But with many common aspects
- Types of
  - Reduced Instruction Set Computer – RISC
  - Complex Instruction Set Computer – CISC

# The MIPS Instruction Set

- Used as the example throughout the book
  - Originated from Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://www.mips.com))
- Large share of embedded core market
- Typical features of many modern ISAs
  - See MIPS Reference Card, and Appendixes B and E



# Arithmetic Operations

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

add a, b, c    #    a = b + c
- All MIPS arithmetic operations have this regular form
- *Design Principle 1: Simplicity favors regularity*
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost



# Arithmetic Example

- C/C++ code:

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

- Compiled pseudo-MIPS assembly code:

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



# Operands in MIPS Assembly

- Register operands
- Memory operands
- Immediate operands (constant)

# Register Operands

- Arithmetic instructions use register operands
- MIPS architecture has a  $32 \times 32$ -bit register file
  - Used for frequently accessed data
  - Numbered 0 to 31
  - Each register is a 32-bit word
- Names recognized by MIPS assembler
  - \$t0~\$t9, \$s0~\$s7, etc.
  - Or \$0-\$31, accepted by certain assemblers
- *Design Principle 2: Smaller is faster*

# Register Operands

- \$zero: constant 0 (reg 0, also written as \$0)
- \$at: Assembler Temporary (reg 1, or \$1)
- \$v0, \$v1: result values (reg's 2 and 3, or \$2 and \$3)
- \$a0 – \$a3: arguments (reg's 4 – 7, or \$4 - \$7)
- \$t0 – \$t7: temporaries (reg's 8 – 15, or \$8 - \$15)
- \$s0 – \$s7: saved (reg's 16 – 23, or \$16 - \$23)
- \$t8, \$t9: temporaries (reg's 24 and 25, or \$24 and \$25)
- \$k0, \$k1: reserved for OS kernel (reg's 26 and 27, \$26/27)
- \$gp: global pointer for static data (reg 28, or \$28)
- \$sp: stack pointer (reg 29, or \$29)
- \$fp: frame pointer (reg 30, or \$30)
- \$ra: return address (reg 31, or \$31)



# Register Operand Example

- C/C++ code:

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

- Put  $f$ ,  $g$ ,  $h$ ,  $i$ , and  $j$  in  $\$s0$ ,  $\$s1$ ,  $\$s2$ ,  $\$s3$ , and  $\$s4$ , respectively

- Compiled MIPS code:

`add $t0, $s1, $s2`

`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

# Memory Operands

- Memory used mainly for composite data
  - Arrays, structures, dynamic data
- In arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- MIPS memory is byte addressable
  - Each address identifies an 8-bit byte
- Memory is organized in words
  - Word address must be a multiple of 4 – alignment restriction
- MIPS is Big Endian (except some MIPS extension)
  - Most-significant byte at least address of a word
  - Little Endian: least-significant byte at least address



# Memory Operand Example 1

- C/C++ code:

`g = h + A[8];` //g, h, A are words

- g in \$s1, h in \$s2, base address of A in \$s3

- Compiled MIPS code:

- Index 8 requires offset of 32 (4 bytes/word)

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

offset

base address 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
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!



# Immediate Operands

- 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 loading data from memory



# 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

# 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

sll/srl rd, rt, shamt

- rt: source register
- rd: destination register
- shamt: how many bits to shift
- Shift left logical
  - Shift left and fill vacated bits with 0 bits
  - sll by  $i$  bits = multiplies by  $2^i$
- Shift right logical
  - Shift right and fill vacated bits 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 doesn't have NOT instruction, implemented with NOR instruction

`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

# Load 32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant  
`lui rt, constant`
  - Copies 16-bit constant to left 16 bits of `rt`
  - Clears right 16 bits of `rt` to 0

`lui $s0, 61`

0000 0000 0011 1101	0000 0000 0000 0000
---------------------	---------------------

`ori $s0, $s0, 2304`

0000 0000 0011 1101	0000 1001 0000 0000
---------------------	---------------------





# Branch/Jump 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:

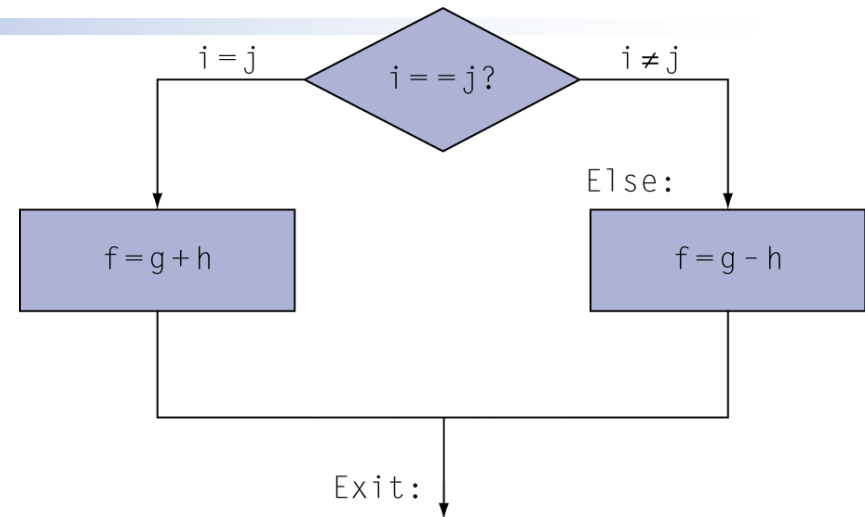
```
if (i==j) f = g+h;  
else f = g-h;
```

- f,g,h,i,j in \$s0, \$s1, \$s2, \$s3, \$s4 respectively

- 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;
```

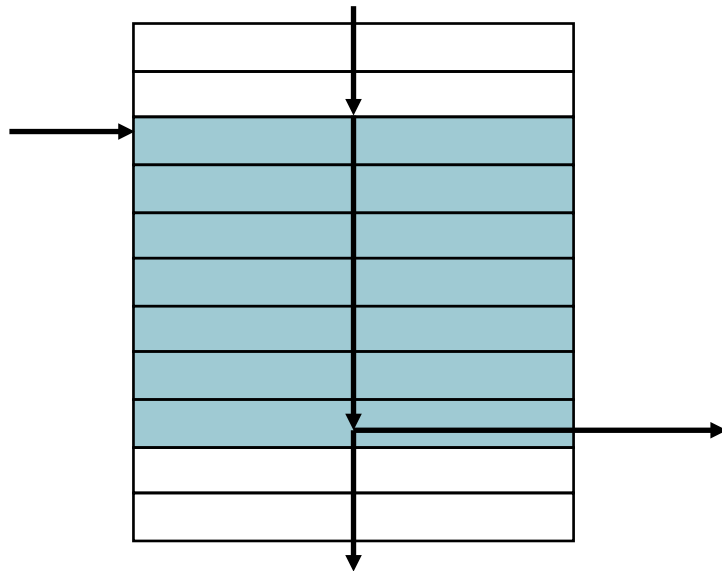
- i in \$s3, k in \$s5, address of save in \$s6

- Compiled MIPS code:

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

# Basic Blocks

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



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

# 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 ( $rs < \text{constant}$ )  $rt = 1$ ; 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 b1t, bge, etc?
- Hardware for  $<$ ,  $\geq$ , ... slower than  $=$ ,  $\neq$ 
  - Combining branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common cases
- This is a good design compromise

# Assembler Pseudoinstructions

- Most assembly instructions and machine instructions have one-to-one correspondence
- Pseudoinstructions: not a real implementation, assembler's imagination

`move $t0, $t1`       $\rightarrow$    `add $t0, $t1, $zero`

`blt $t0, $t1, L`     $\rightarrow$    `slt $at, $t0, $t1`  
                                 `bne $at, $zero, L`

- `$at` (register 1): assembler temporary

# Benchmark Programs

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC CPU2006 INT	SPEC CPU2006 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%