CPE 431/531

Chapter 2 – Instructions: Language of the Computer

Dr. Rhonda Kay Gaede

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

2.1 Introduction

- The words of a computer's language are called instructions and its vocabulary is called an instruction set architecture.
- Instruction sets are more similar than they are different, however there are two camps:
  - RISC – MIPS, this
  - CISC – 68000, IA

2.2 Basics of MIPS Arithmetic

- We need arithmetic

  `add a, b, c`

  $a \leftarrow b+c$

  add b,c

  $b \leftarrow b+c$

- From high level

  `a = b + c + d + e;`

- Fixing the number of operands keeps the hardware simple.
- Design Principle 1: Simplicity favors regularity

```
add  a,b,c
add  a,a,d
add  a,a,e
```

I=three
no extra storage

```
add  a,b,c
add  f,d,e
add  a,a,f
```

I=3
extra storage

```
add c,d,e
add a,b,c
```

2.2 Compiling C into MIPS

Compilation is the process of creating MIPS assembly language from a high level language.

Examples:

```
(1) a = b + c;
(2) d = a - e;
(3) f = (g + h) - (i + j);
```

```
(1) add  a, b, c
(2) sub  d, a, e
```

```
(3) add  $t0, g, h
    add  $t1, i, j
    sub  f, $t0, $t1
```

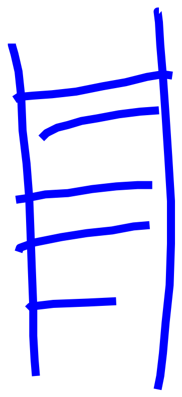
```
add  f, g, h
sub  f, f, i
sub  f, f, j
```

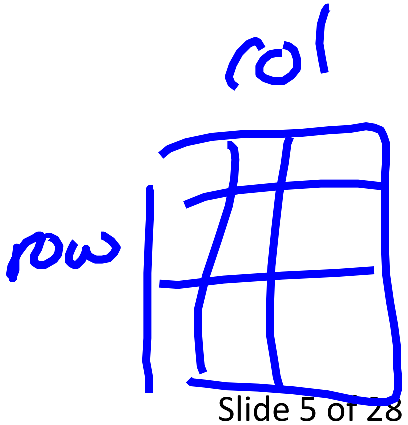
$2^{10} = 1024$

2.3 MIPS Basics

- In high level languages, variables live in _memory_
- In MIPS assembly, operands live only in _registers_
- _Data transfer_ instructions move variables from memory/registers to registers/memory.
- MIPS has _32_ registers and an address space of _2^{32}_ memory bytes.
- Design Principle 2: Smaller is faster.

Decoding takes longer when the number of registers increases.
More registers means that instruction fields must be larger.





2.2 Compiling C into MIPS (Registers)

Reconsider

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

$s0 -
save
registers

The variables **f, g, h, i,** and **j** are assigned to registers **$s0, $s1, $s2, $s3,** and **$s4,** respectively.

$t0 -
temporary
registers

```
add  $s0, $s1, $s2
sub  $s0, $s0, $s3
sub  $s0, $s0, $s4
```

2.3 Memory Operands: First Pass

- Data transfer instructions

  Load   register ← memory

  Store   memory ← register

- Compiling an Assignment When an Operand is in Memory

  The compiler has associated g with **$s1** and h with **$s2** and A is an array of 100 words, base pointer **$s3**

  g = h + A[8];

  $\boxed{\text{lw}}$ $t0, 8($s3)

  add $s1, $s2, $t0 $t0 ← Mem[$s3+8] ×4

  g h A[8]

- Hardware/Software Interface
  - A compiler translates, associates variables with registers, allocates memory to data structures.

2.3 Memory Operands: Second Pass

- Bytes/Words
  - 32-bit words consist of 4 8-bit bytes
  - Computers are bigendian or little endian depending on whether the first byte is most or least significant
  - MIPS is byte addressable
- Compiling Using Load and Store

  h is associated with **$s2** and the base address of **A** is in **$s3**

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

```
lw   $t0, 32($s3)
add  $t0, $t0, $s2
sw   $t0, 48($s3)
```

$t0 ← A[8]

- . Hardware/Software Interface
  - The compiler keeps frequently used items in registers, spills other variables to memory.

2.3 Constant or Immediate Operands

- More than half of the MIPS arithmetic instructions have a _constant_ as an operand when running the SPEC CPU 2006 benchmarks.
- With the instructions we've seen so far, constants must be put in _memory_ when the program was loaded and then we would have to load them into a _register_ to use.
- The alternative is to add a different kind of instruction.

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

- Design Principle 3: Make the common case fast.

addi a, b, 4

2.4 Signed and Unsigned Numbers

- Unsigned

  1011_2 = B, 11 $1\times2^3+0\times2^2+1\times2^1+1\times2^0$

  Range for n bits: $0 - 2^n - 1$

- Signed

  1011_2 = $-1\times2^3+0\times2^2+1\times2^1+1\times2^0 = -5$

  Range for n bits: -2^{n-1} – $+2^{n-1}-1$

- Finding the 2's complement

  10 00001010 ← 1111 0101 + 0000 0001 1111 0110

  11110110

  00101000

- Sign Extension

  00001010 0000 0000 0000 1010

  10110111 1111 1111 1011 0111

0009
2699

2.5 R-type Instruction Format

- Translating a MIPS assembly instruction into a machine instruction:

  `add $t0, $s1, $s2` (R-type)

Op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
0	17	18	8	0	32
000000	10001	10010	01000	00000	10 0000

0x 0232 4020

- MIPS Fields

op:	opcode
rd:	register destination
rs:	first register source
rt:	second register source/destination register for lw
shamt:	shift amount
funct:	function code

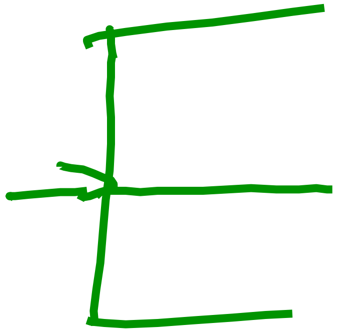
2.5 I-Type Instruction Format

- One size doesn't fit all. `lw` and `sw` have different requirements than `add`.
- Design Principle 4: Good design demands good compromises.

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

op(6)	rs(5)	rt(5)	constant or address(16)
100011	10011	01000	0000 0000 0010 0000

- For data transfer, address offset is limited to $\pm 2^{15}$ bytes, $\pm 2^{13}$ words.
- Another Translation Example: A[300] = h + A[300];



2.5 Instructions for Making Decisions

- Two conditional ones for now:

```
beq register1, register2, L1
```

```
if (register1 == register2) then
        PC ← PC+4 +L1*4
else
        PC ← PC+4
```

```
bne register1, register2, L1
```

neq case

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

```
(1) beq  $s3, $s4, Then
(2) sub  $s0, $s1, $s2
(3) j    Exit
Then: add $s0, $s1, $s2
(4) Exit:  next
```

neq case

```
(1) bne  $s3, $s4, Else
    add  $s0, $s1, $s2
    j    Exit
(2) Else: sub $s0, $s1, $s2
(3) Exit:  next
```

2.7 Adding less than or greater than

- Less than is useful, i.e., for (i = 0; i < 10; i++)

```
slti $t0, $s1, 10
```

```
if($s1 < 10) then
    $t0 ← 1
else
    $t0 ← 0
```

```
bne $t0, $zero, offset
```

register that always is = 0

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

```
if ($s0 < $s1) then
        $t0 ← 1
else
    $t0 ← 0
```

$a < b$

$b > a$

2.7 Compiling a while loop

Consider

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

where i is associated with `$s3` and `k` with `$s5` and the base of array `save` is `$s6.`

```
Loop:  sll   $t1, $s3, 2           shift left
       add   $t1, $t1, $s6            logical
       lw    $t0, 0($t1)  $t0 <= save[i]
       bne   $t0, $s5, Exit
       addi  $s3, $s3, 1           i++
       j     Loop
Exit:
```

$t0 <- Mem[$t1+0]

2.8 Supporting Procedures in Computer Hardware

- Steps involved in calling a procedure (function)
  1) Make parameters available to the called procedure
  2) Transfer control to the procedure
  3) Acquire the needed space for the procedure.
  4) Perform the desired task
  5) Make result available to the calling procedure
  6) Transfer control back to calling procedure
- Support comes in registers and instructions
  - Registers
    - `$a0-$a3` – used to pass parameters in
    - `$v0-$v1` – used for return values
    - `$ra` return address
  - Instructions
    - `jal`, $ra ← PC+4, PC ← Procedure Address
    - `jr` PC ← $ra

2.8 Compiling a Leaf Procedure

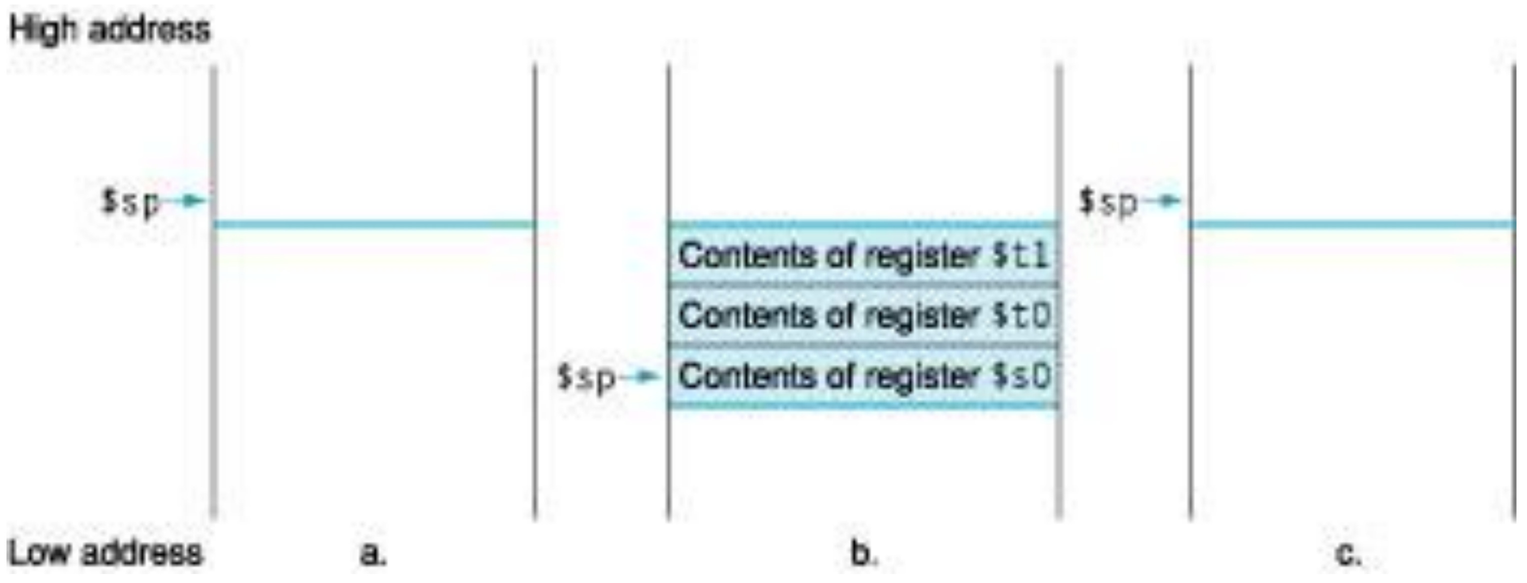
```
int leaf_example (int g, int h, int i, int j~~)~~, int k)
{
     int f;
     f = (g + h) - (i + j);
     return(f);
}

leaf_example:  ~~sub      $sp, $sp, 12~~ 8
               ~~sw       $t1, 8($sp)~~
               ~~sw       $t0, 4($sp)~~
               ~~sw       $s0, 0($sp)~~
               add      $t0, $a0, $a1
               add      $t1, $a2, $a3
               sub   v0 ~~$s0~~, $t0, $t1
               ~~add      $v0, $s0, $zero~~
               ~~lw       $s0, 0($sp)~~
               ~~lw       $t0, 4($sp)~~
               ~~lw       $t1, 8($sp)~~
               ~~add      $sp, $sp, 12~~ 8
               jr       $ra
```

same as move →

$t0 ← g+h
$t1 ← i+j
$s0 ← $t0 - $t1

2.8 Leaf Example Stack

low



high

- In the previous example, what happens if we change the procedure to have one more argument? _Spill them to the stack_

2.8 Nested Procedures

- Calling Procedure
  - Pushes its argument registers onto the stack so it can put arguments there for the callee
  - Pushes any temporary registers it needs after the call onto the stack
  - Pushes **`$ra`** onto the stack
- Called Procedure
  - Pushes saved registers it plans to use onto the stack

2.8 Nested Procedure Compilation

? caller

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

n=3

a0←3

```
200 fact:  addi   $sp, $sp, -8       }
204        sw     $ra, 4($sp)        } preparing for call
208        sw     $a0, 0($sp)        }
212        slti   $t0, $a0, 1        }
216        beq    $t0, $zero, L1     } $a0 < 1? a0←2
220        addi   $v0, $zero, 1      $a0<1
224        addi   $sp, $sp, 8
228        jr     $ra
232 L1:           addi   $a0, $a0, -1
236        jal    fact
240        lw     $a0, 0($sp)        }
244        lw     $ra, 4($sp)        }
248        addi   $sp, $sp, 8        } clean up
252        mul    $v0, $a0, $v0      }
256        jr     $ra
```

place on stack

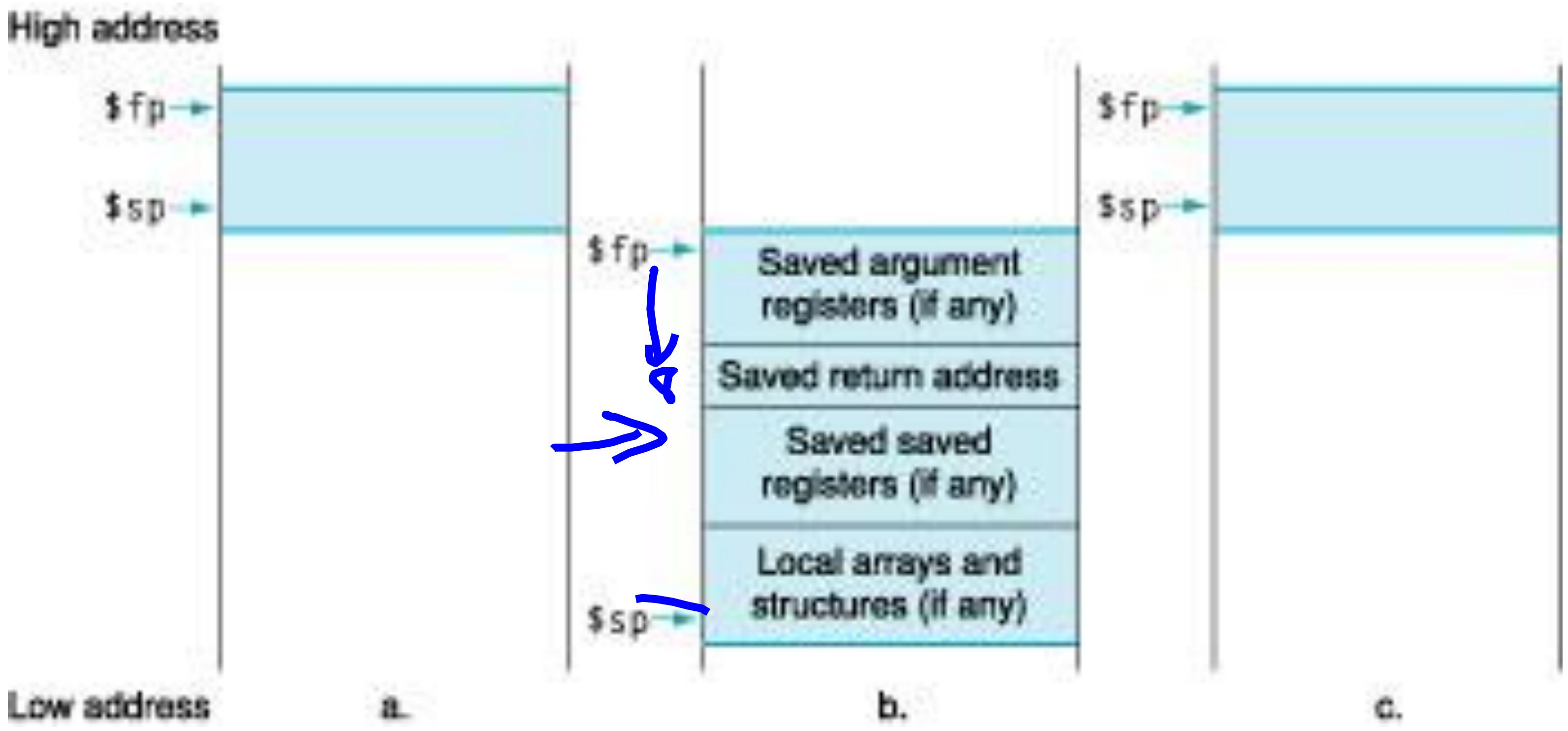
PC

200
204
208
212
216
232
236
200
204
208
212
216
232
a0←1 236
200
204
208
212
216
232
a0←0 236

200
204
208
212
216
$v0←1 220
224
228
240
244
a0←1 248
ra←240 252
v0←1 256
240
244
a0←2 248
ra←240 252
v0←2 256
240
244
a0←3 248
ra←? 252
v0←6 256

2.8 More About the Stack

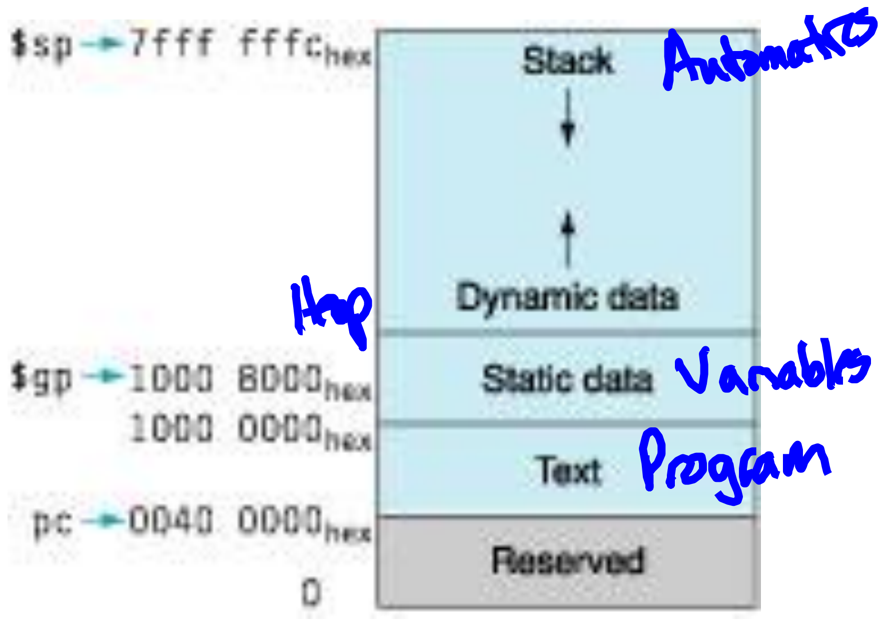
- Allocating Space for Automatic Variables
  - In addition to storing saved registers, the stack holds local variables that don't fit into registers, e.g., arrays, structs.
  - Saved registers + Local Variables = Procedure Frame


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.

2.8 The Heap

- Space is needed for static variables and dynamic data structures
  - Space is reserved and freed on the heap using explicit system calls.
- Register Usage

`$zero`	0
`$v0-$v1`	2-3
`$a0-$a3`	4-7
`$t0-$t7`	8-15
`$s0-$s7`	16-23
`$t8-$t9`	24-25
`$gp`	28
`$sp`	29
`$fp`	30
`$ra`	31



2.10 32-bit Immediate Operands

op	rs	rt	immediate
6	5	5	16

- 32-Bit Immediate Operands
  - Upper 16 Bits - **`lui`**
  - Lower 16 Bits - **`ori`**

lui $t0, 61

$t0 ← 61 << 16

- Loading `0x003D 0900`

lui $s0, 61

ori $s0, $s0, 2304

$s0 0x 003D 0000

addi

sign extend

0x 003D F900

$\boxed{003D}$ 0000

~~FFFF~~ F900

ori

003D 0000

0000 F900

2.10 32-Bit Addresses

- Addresses in Branches and Jumps

```
j 10000
```

j
opcode 6 | address 26

$PC \leftarrow$ address << 2

```
bne $s0, $s1, Exit
```

  – Elaboration: For jumps, we give only 28 bits, from whence springeth the other 4?

PC+4

beq $s1, $s2, L1

  – Branching Far Away

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

jr $s0
$PC \leftarrow$ $s0

2.11 Parallelism and Instructions: Synchronization

- Cooperation between tasks usually means some tasks are _writing_ new values that others must _read_
- In computing, synchronization mechanisms are typically built with _user-level_ software routines that rely on _hardware_-supplied synchronization instructions
- One hardware primitive will both read from and write to a location in one _atomic_ operation
- The other approach is to have a _pair_ of instructions in which the _second_ instruction _returns_ _a_ _value_ showing whether the pair of instructions was executed as if the pair were atomic
- MIPS has _load_ _linked_, _ll_, and _store_ _conditional_, _sc_

ll

2.11 Code Sequence for Atomic Exchange

```
again: addi $t0,$zero,1        ;copy locked value
       ll   $t1,0($s1)         ;load linked
       sc   $t0,0($s1)         ;store conditional
       beq  $t0,$zero,again    ;branch if store fails
       add  $s4,$zero,$t1      ;put load value in $s4
```

1 locked
∅ unlocked

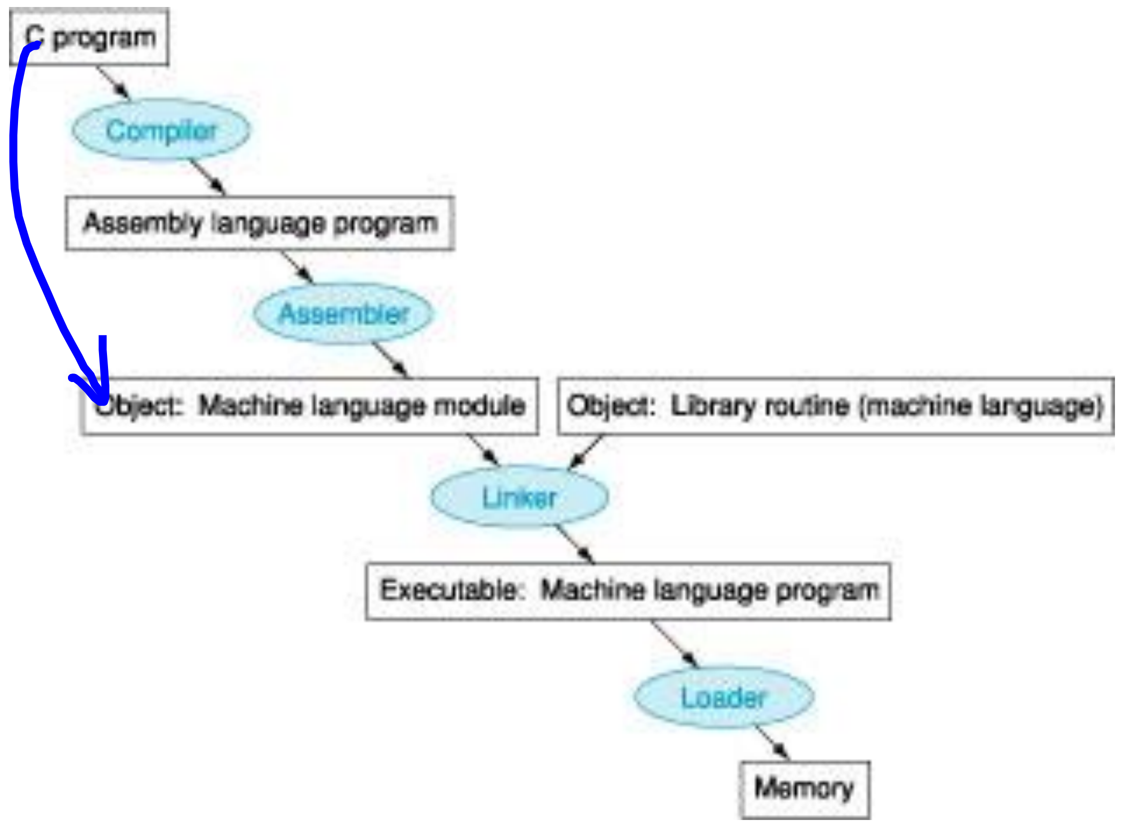
This can change $t∅ →

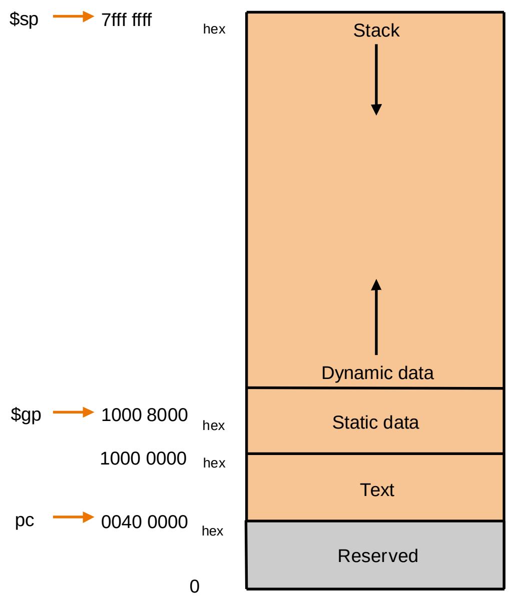
$$\$t\emptyset \leftarrow 1$$
$$\$t1 \leftarrow \text{Mem}[\$s1+\emptyset]$$

sc
```
{ Mem[$s1+∅] ← 1 ($t∅)
  If Mem[$s1+0] hasn't changed
        $t∅ ← 1
  else
        $t∅ ← 0 }
```

try
throw
catch

2.12 Translating and Starting a Program


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory



2.20 Fallacies and Pitfalls

- Fallacy: More powerful instructions mean higher performance.
- Fallacy: Write in assembly language to obtain the highest performance.
- Fallacy: The importance of commercial binary compatibility means successful instruction sets don’t change.
- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.
- Pitfall: Using a pointer to an automatic variable outside its defining procedure.