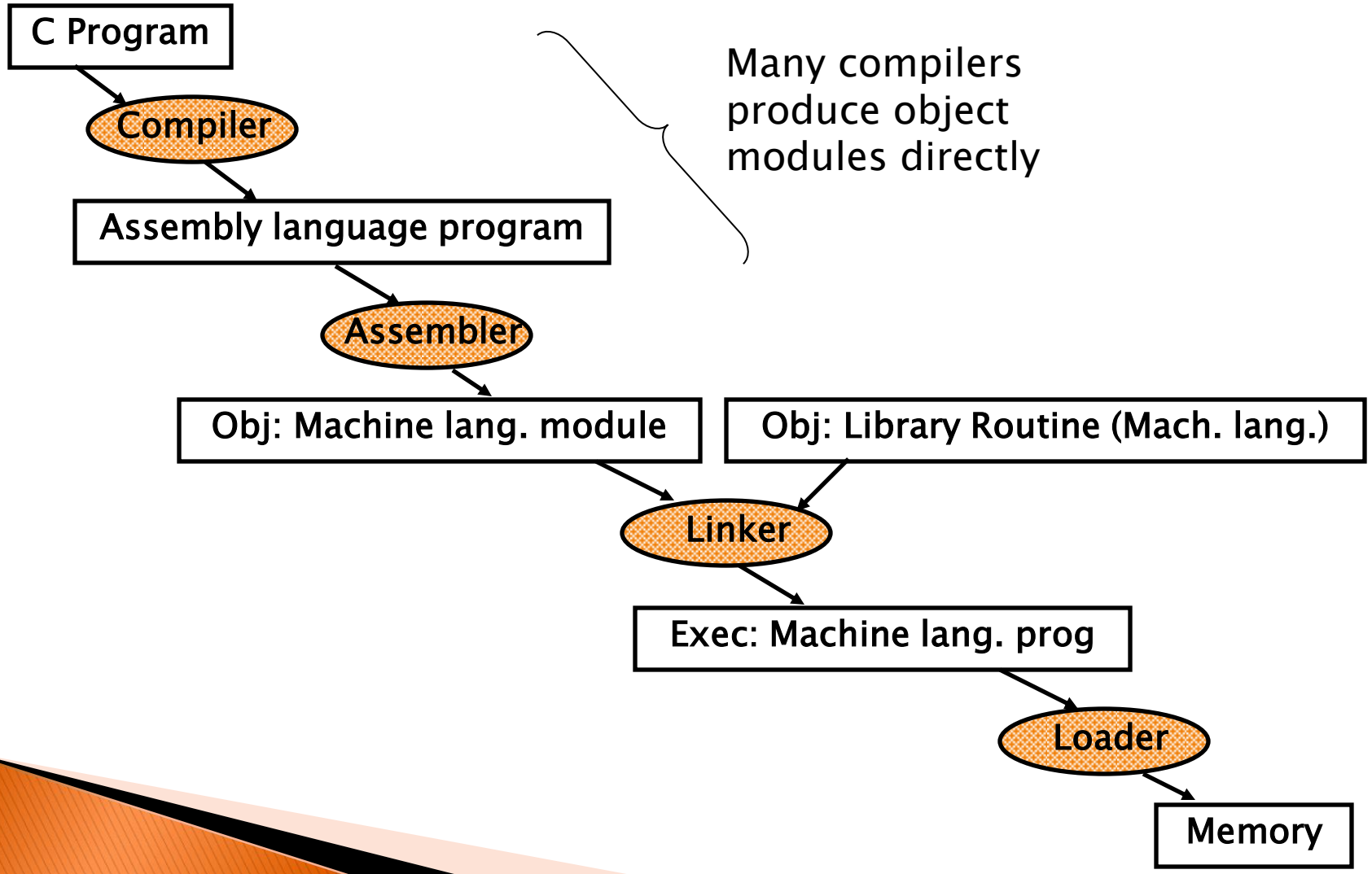
CSE 331
Computer Organization

Lecture 3
Assembly Programs

A Translation Hierarchy



Advantages & Disadvantages

- Assembly programming is useful when the speed or size of a program is important.
- But assembly languages are machine specific and they must be rewritten to run on another machine.
- Another disadvantage is that assembly language programs are longer than the equivalent programs written in a high-level language.
- It is also true that programs written in assembly are more difficult to read and understand and they may contain more bugs.

Assembly Language & Programming

- **Assembly language** is the symbolic representation of a computer's binary encoding, which is called **machine language.**
- Assembly language is more readable than machine language because it uses symbols instead of bits.
- Assembly language permits programmers to use ***labels*** to identify and name particular memory words that hold instructions or data.
- A tool called ***assembler*** translates assembly language into binary instructions.
- An assembler reads a single assembly language *source file* and produces *object file* containing machine instructions and bookkeeping information that helps combine several object files into a program.

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

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

- Produces an executable image

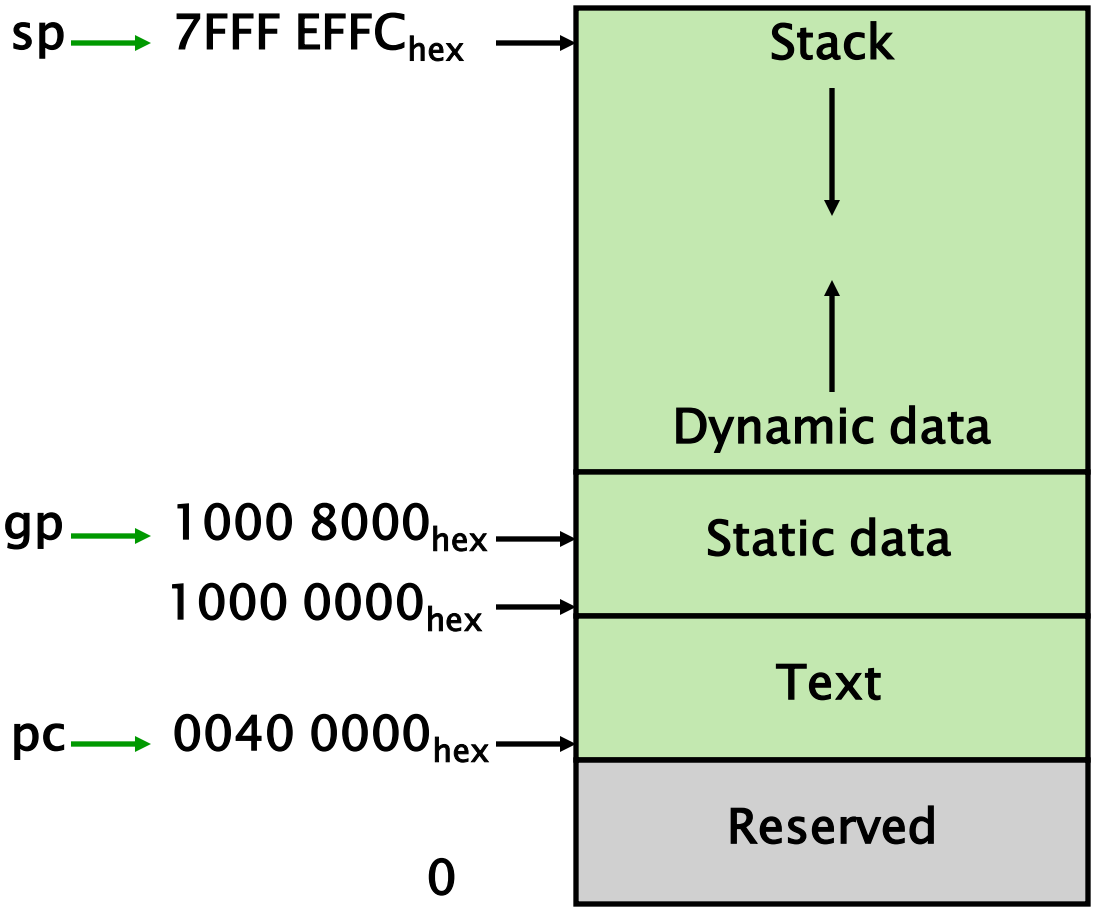
1. Merges segments
2. Resolve labels (determine their addresses)
3. Patch location-dependent and external refs

Loading a Program

- Load from image file on disk into memory

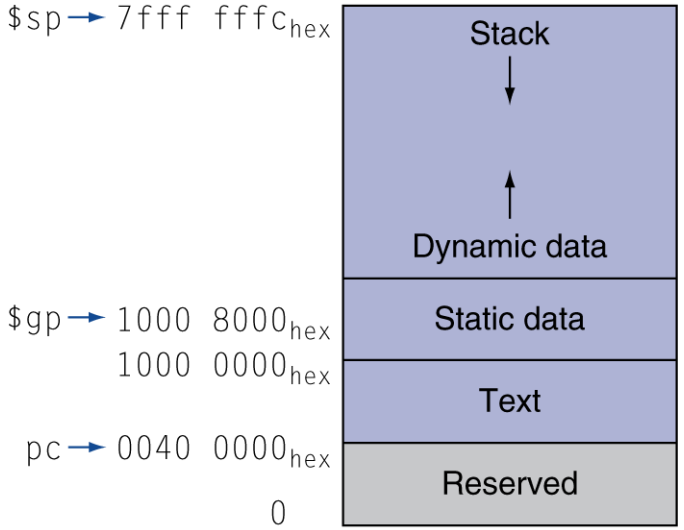
1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text and initialized data into memory
   - Or set page table entries so they can be faulted in
4. Set up arguments on stack
5. Initialize registers (including $sp, $fp, $gp)
6. Jump to startup routine
   - Copies arguments to $a0, … and calls main
   - When main returns, do exit syscall

The MIPS Memory Allocation for Program & Data



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
- Stack: automatic storage



Two Obj. Files

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Executable

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Assembler Syntax

- Comments in assembler files begin with a sharp sign (#)
- Instruction opcodes (e.g. lw, add, etc.) are reserved words that cannot be used as identifier.
- Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```
        .data
item:   .word 25
        .text
        .globl main    # must be global
main:   lw $t0, item
        . . .
```

Assembler Syntax (Cont'd)

- Numbers are base 10 by default, for example:

```
addi $t0, 20   ⟹   addi $t0, 0x14
```

- Strings are enclosed in double quotes (")

```
        .data
temp1:
        .word 3
str:
        .asciiz "Result = "
        .text
        .globl main
main:
        lw  $a0, temp1        # load temp1 into a0
        . . .
```

Two Similar Examples

```
        .text
        .globl main
main:
        li      $t0, 0x0005       # put 5 into register t0
        li      $t1, 0x0017       # put 23 into register t1
        add     $t2, $t0, $t1     # t2 ⟵ t0 + t1
```

```
        .data
temp1:
        .word 5
temp2:
        .word 23
        .text
        .globl main
main:
        lw      $t0, temp1        # put 5 into register t0
        lw      $t1, temp2        # put 23 into register t1
        add     $t2, $t0, $t1     # t2 ⟵ t0 + t1
```

System Services

Service	System call code	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
print_double	3	$f12=double	
print_string	4	$a0=string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0=buffer, $a1=length	
sbrk	9	$a0=amount	address (in $v0)
exit	10		

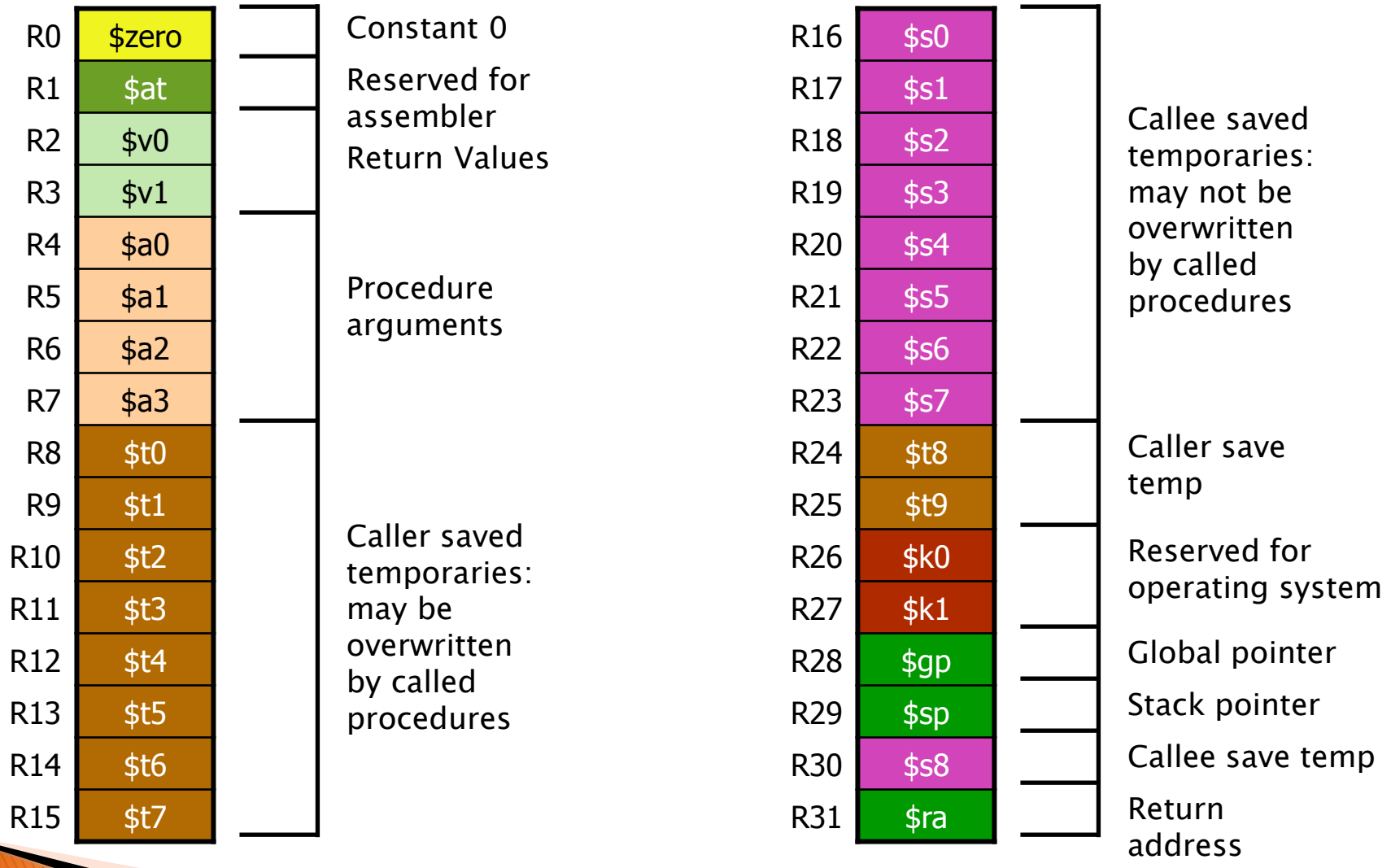
System Call Example

```
   .data
str:
   .asciiz "Result is "
   .text
   .globl main
main:
   li   $v0, 5          # system call code for read_int
   syscall              # read int
   move $t0, $v0        # move integer to t0

   li   $v0, 5          # system call code for read_int
   syscall              # read another int
   move $t1, $v0        # move integer to t1

   add  $t2, $t0, $t1   # add t0 and t1 and put the result into t2

   li   $v0, 4          # system call code for print_str
   la   $a0, str        # address of string to print
   syscall              # print the string

   move $a0, $t2        # copy t2 to a0
   li   $v0, 1          # system call code for print_int
   syscall              # print it
```

MIPS Register Conventions

- Conventions
  - This is an agreed upon "**contract**" or "**protocol**" that everybody follows
  - Specifies correct (and expected) **usage**, and some **naming** conventions
  - **Established part of architecture**
  - Used by all compilers, programs, and libraries
  - Assures **compatibility**

MIPS Register Conventions



Program counter

- We need a register to hold the address of the current instruction being executed
  - "Program Counter" `PC` in MIPS
- jal saves `PC+4` in register $ra
- At the end of the procedure we jump back to the $ra (an unconditional jump)

`jr $ra`

- The caller puts the parameter values in $a0-$a3
- The caller uses `jal X` to jump to procedure X
- The callee performs the calculations, places the results in $v0-$v1
- Returns control to the caller by `jr $ra`

Stack

- Suppose the procedure needs more than 4 arguments
- We store the values in Stack (a last-in-first-out queue)
- A stack needs a pointer to the most recently allocated address in the stack: stack pointer
- Placing data onto the stack is called a Push. Removing data from the stack is called a Pop.
- The stack pointer in MIPS is $sp. By convention stacks "grow" from higher addresses to lower addresses!!! (You push values onto the stack by subtracting from the stack pointer)

Procedure call

- When making a procedure call, it is necessary to
  1. Place inputs where the procedure can access them
  2. Transfer control to procedure
  3. Acquire the storage resources needed for the procedure
  4. Perform the desired task
  5. Place the result value(s) in a place where the calling program can access it
  6. Return control to the point of origin

- MIPS
  - Provides instructions to assist in procedure calls (jal) and returns (jr)
  - Uses software conventions to
    - place procedure input and output values
    - control which registers are saved/restored by caller and callee
  - Uses a software stack to save/restore values

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

Simple Procedure Call

```
   .text
   .globl get_square

get_square:
   mult  $a0, $a0
   mflo  $v0
   jr    $ra

   .data
temp1:
   .word 3
str:
   .asciiz "Result = "

   .text
   .globl main
main:
   lw    $a0, temp1          # load temp1 into a0
   jal   get_square          # save address of next instr. into ra register
                             # and jump to get_square procedure
   move  $t0, $v0            # put the result in t0

   li    $v0, 4              # system call code for print_str
   la    $a0, str            # address of string to print
   syscall                   # print the string

   li    $v0, 1              # system call code for print_int
   move  $a0, $t0            # copy result to a0
   syscall                   # print it
```

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

- 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

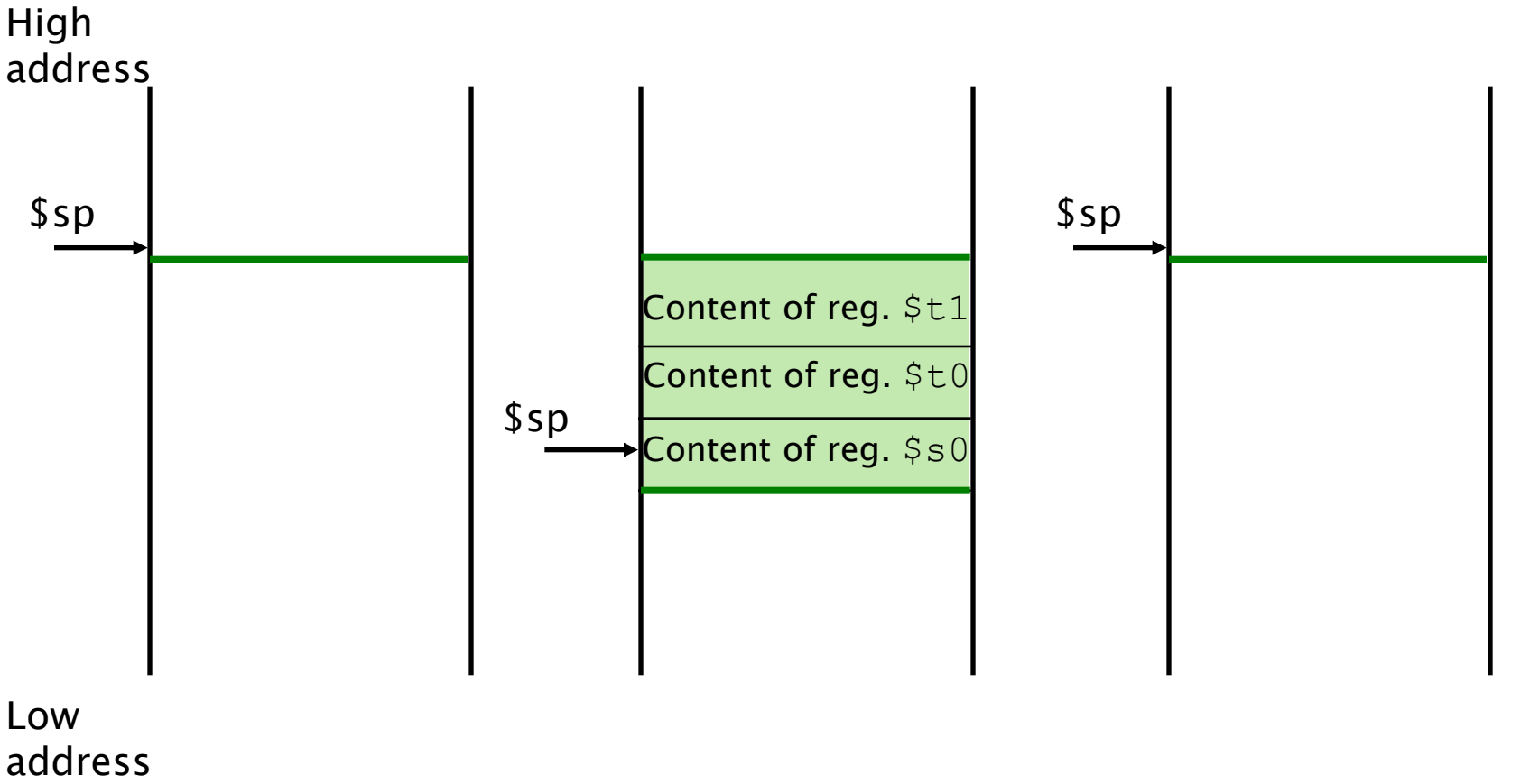
A procedure call with a stack

```
int leaf-example (int g, int h, int i, int j)
{
    int f;
    f = (g+h)-(i+j);
    return f;
}
```

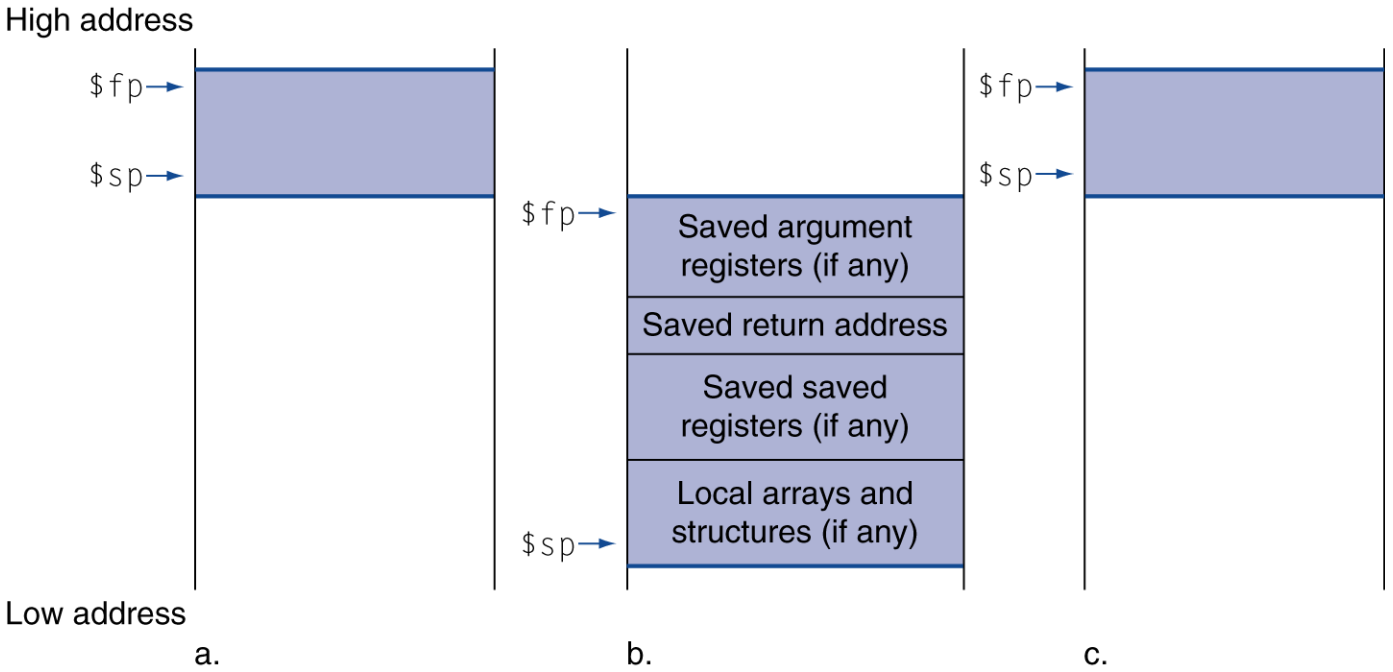
Assume the parameter variables g, h, i, and j correspond to the argument registers $a0, $a1, $a2, and $a3, and f corresponds to $s0.

```
leaf_example:
        addi    $sp, $sp, -12       # adjust stack to make room for 3 items
        sw      $t1, 8($sp)         # save register $t1 for use afterwards
        sw      $t0, 4($sp)         # save register $t0 for use afterwards
        sw      $s0, 0($sp)         # save register $s0 for use afterwards
        add     $t0, $a0, $a1       # register $t0 contains g + h
        add     $t1, $a2, $a3       # register $t1 contains i + j
        sub     $s0, $t0, $t1       # register $s0 contains (g + h) - (i + j)
        add     $v0, $s0, $zero     # register $v0 contains the result
        lw      $s0, 0($sp)         # restore register $s0 for caller
        lw      $t0, 4($sp)         # restore register $t0 for caller
        lw      $t1, 8($sp)         # restore register $t1 for caller
        addi    $sp, $sp, 12        # adjust stack to delete 3 items
        jr      $ra                 # jump back to calling routine
```

The Values of Stack Pointer & Stack



Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

C Sort Example

- Illustrates use of assembly instructions for a C 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 (Insertion) Sort Procedure in C

```
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, k 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
```

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Nested Procedure Example

- C code:

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

  - Argument n in $a0
  - Result in $v0

Nested Procedure Call Example (Cont'd)

```
   .data
str:
   .asciiz     "The result is "

   .text
   .globl main
main:
   li   $v0, 5          # System call code for read_int
   syscall              # Read int
   move $a0, $v0        # Move integer to $a0

   jal  fact            # Call factorial function
   move $a1,$v0         # Move fact result to $a1

   li   $v0, 4          # System call code for print_str
   la   $a0, str        # Address of string to print
   syscall              # Print the string

   li   $v0, 1          # System call code for print_int
   move $a0, $a1        # Copy result to $a0
   syscall              # Print int

   li   $v0, 10         # System call code for exit
   syscall              # Exit
```

Nested 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 return1
```

Arrays versus Pointers

```
void clear1(int array[ ], int size)
{
    int i;
    for (i = 0;  i < size;  i++);
            array[i] = 0;
}
```

```
void clear2(int *array, int size)
{
    int *p;
    for (p = &array[0];  p< &array[size];  p++);
            *p = 0;
}
```

Example: Clearing and Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

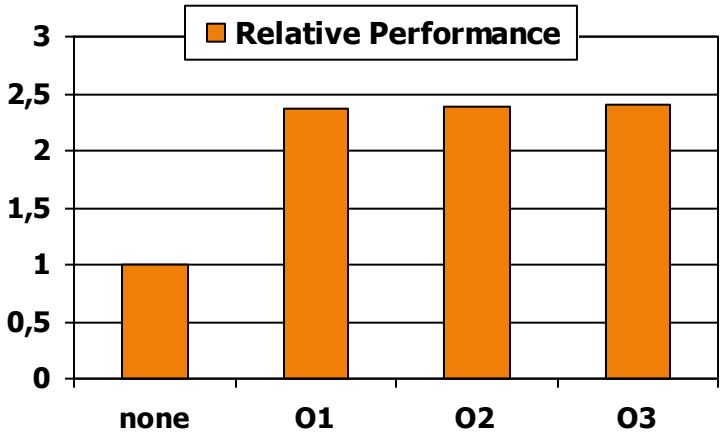
```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

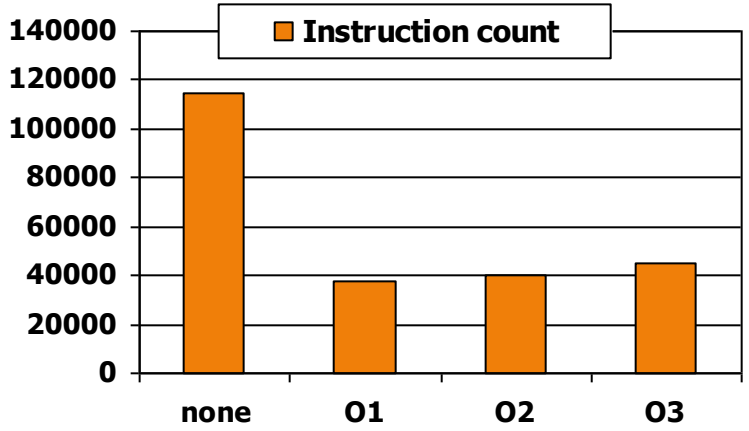
```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

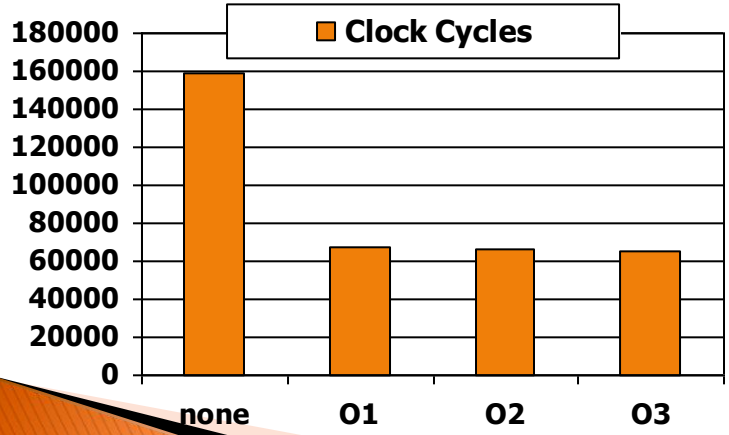
```
       move $t0,$a0    # p = & array[0]
       sll $t1,$a1,2   # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                       #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4  # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                       #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

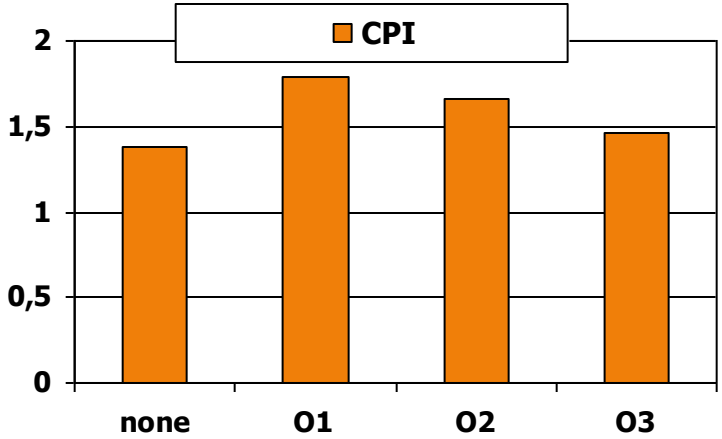
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux









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!

ARM & MIPS Similarities

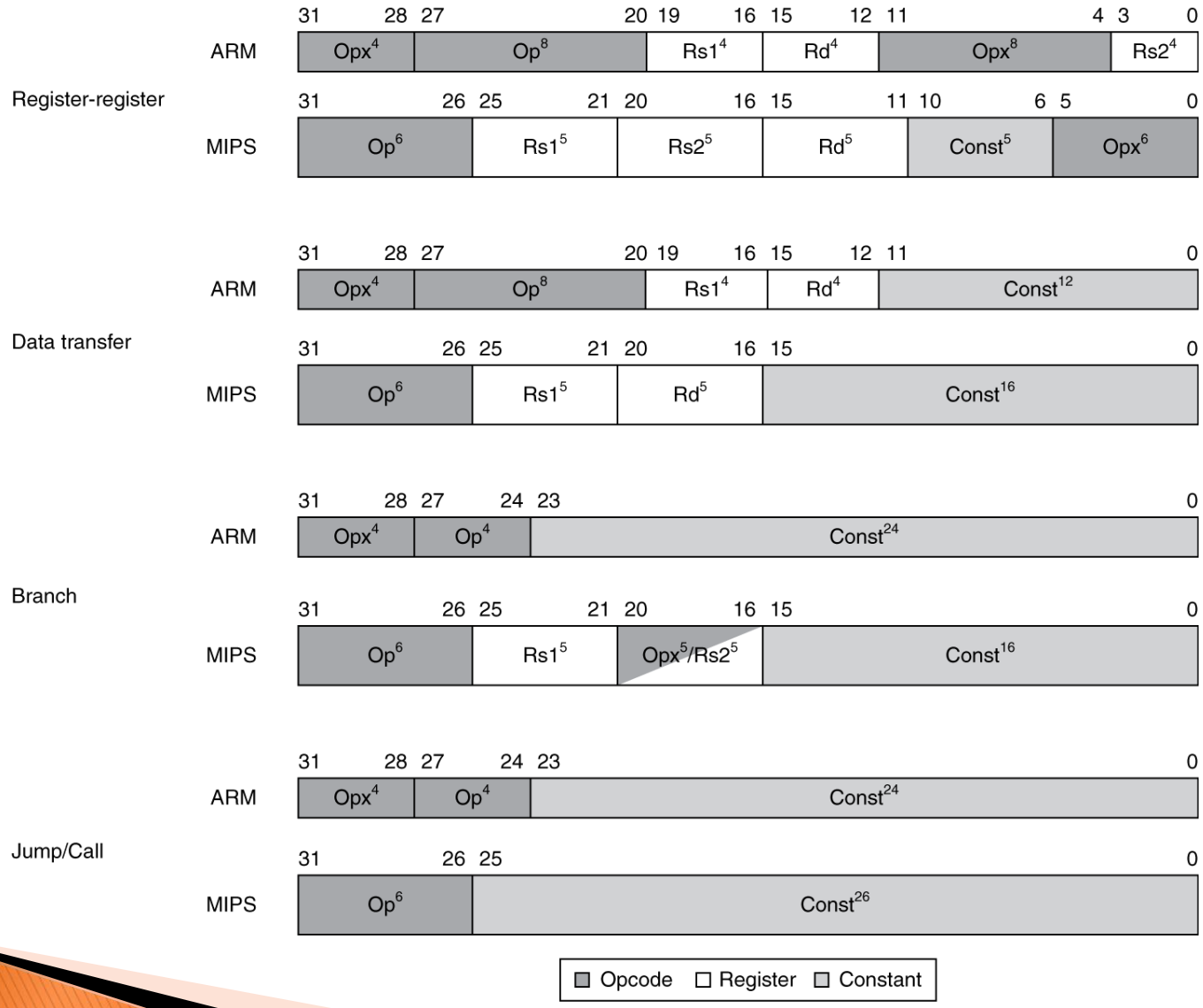
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	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

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - 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

Instruction Encoding

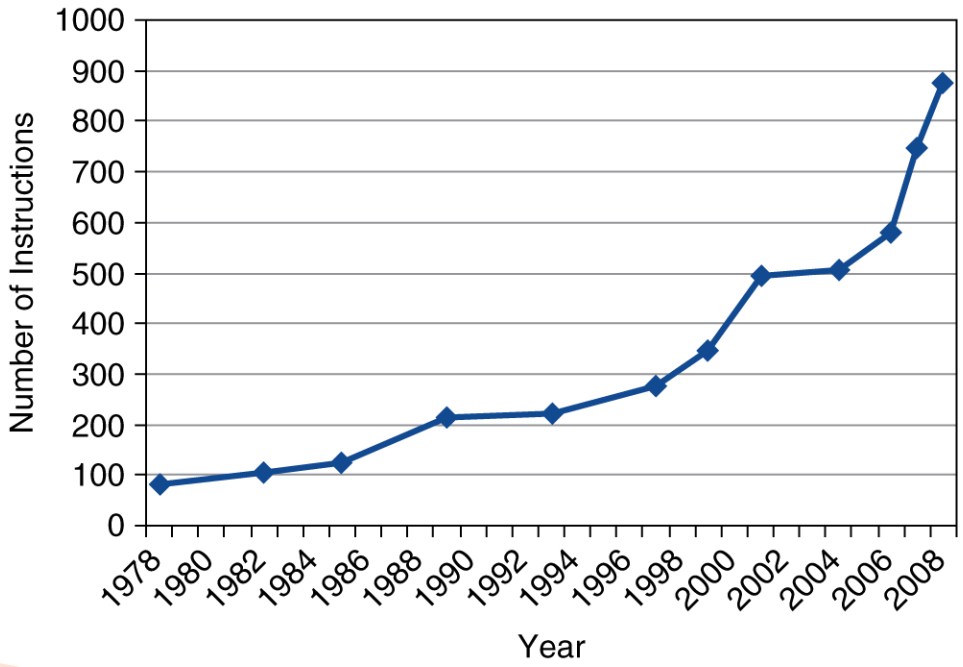


Fallacies

- 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