CSEE 3827: Fundamentals of Computer Systems, Spring 2022

Lecture 7

Prof. Dan Rubenstein (danr@cs.columbia.edu)

Agenda (P&H 2.1-2.8, 2.10)

- Instruction Set Architecture (RISC / MIPS)
- CPU + Memory in MIPS overview
- CPU
  - conceptual view of control
  - registers in the register file
- Memory
  - Addressing in MIPS (multiple-of-4 rule)
- Instruction Types
  - Memory Movement, Data Manipulation, Jump and Branch
- Instruction Formats
- Memory Organization: Heap and Stack
- Procedure Calls

New Book = some same stuff, new notation!!

RISC & MIPS

RISC machines

- RISC = **R**educed **I**nstruction **S**et **C**omputer

- MIPS (Microprocessor without Interlocked Pipeline Stages) is a specific RISC architecture we use from now on (used in the P&H book)

- MIPS (and other RISC architectures) are “load-store” architectures, meaning all operations performed only on operands in registers**.** (The only instructions that access memory are loads and stores)

- Alternative to CISC (Complex Instruction Set Computer) where operations are significantly more complex.

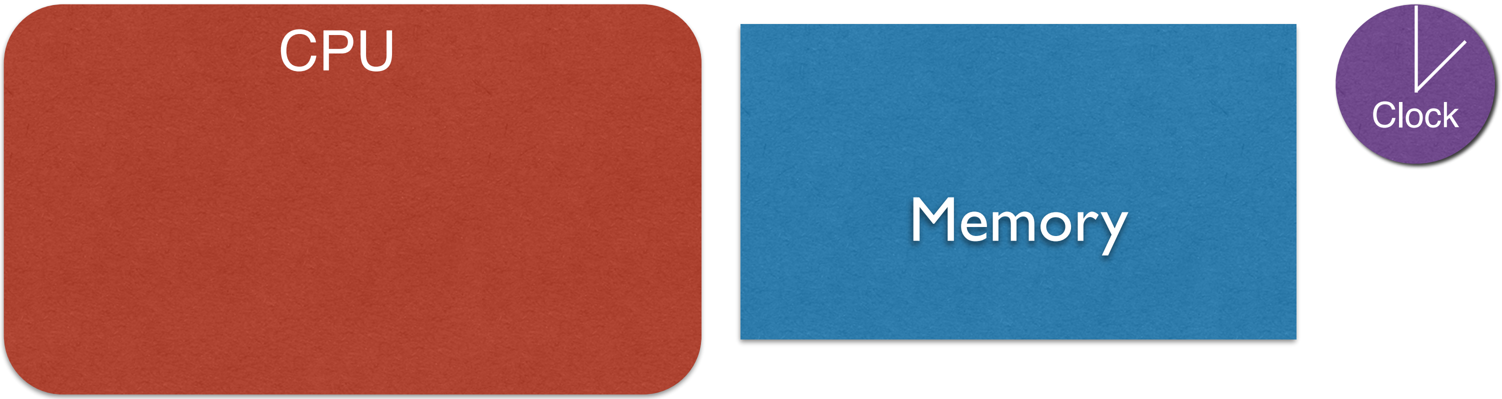
“Parts” of a program from a code perspective

```
while (save[i] == k)
    i += 1;
```

C code

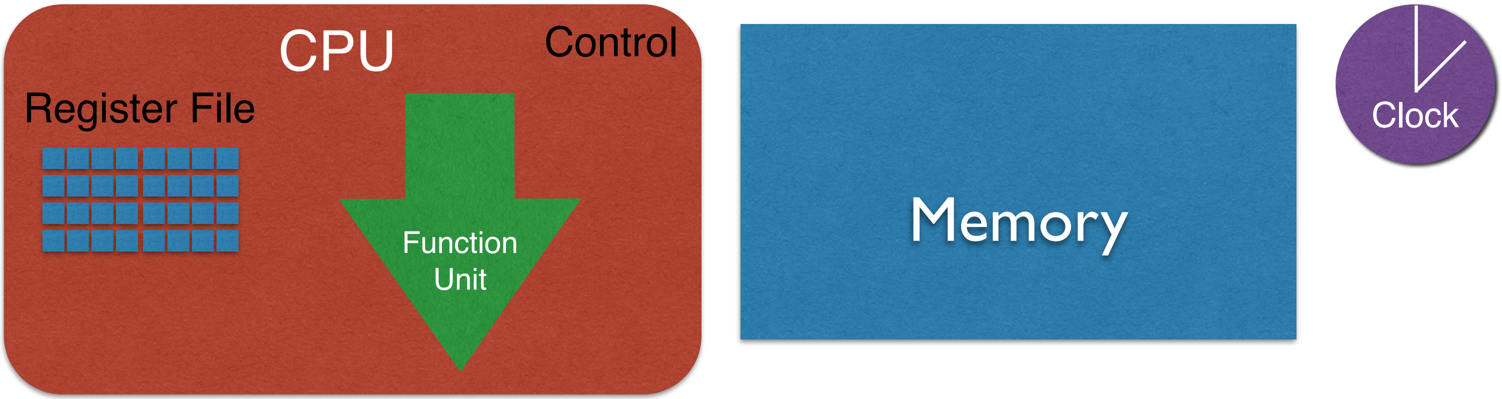
- C code has various “parts” that play different roles
  - “State”: variable i, k, array save[ ], also stores the C-code itself
  - “Operation”:
    - compare values to make decisions on what to do next (save[i] == k)
    - make a change to some “state”: i += 1
  - “Control”: “while” command says “stay in this loop”. Also if/then, for, etc.
- Low level (assembly/machine code) also has these various “parts”, and using them requires a lot more familiarity and involvement

Computer Hardware (from assembly code perspective)



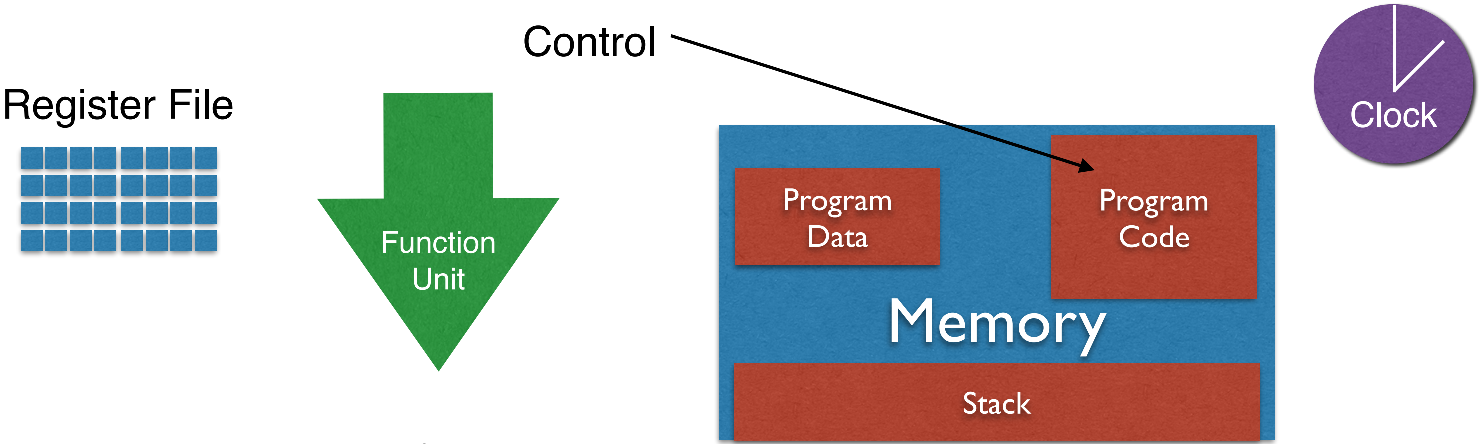
- Hardware consists of:
  - CPU: Control Processing Unit: The “brain” of the computer (state, control, perform operations)
  - Memory: (State only) stores program and additional (program) data
  - Clock: creates clock cycles that the computer uses to execute programs (i.e., in MIPS, an instruction takes a clock cycle to complete)

Computer Hardware (from assembly code perspective)



- CPU:
  - Control: tracks instruction (i.e., line of code to execute) for current clock cycle
  - Function Unit: input → function unit → output (output “lost” as soon as inputs changed, no storage)
  - Register File: collection of registers (state holding data actively worked on)
    - values in registers fed into function unit (as input)
    - results out of function unit stored back in registers (output)
    - Registers maintain values until changed (over several clock cycles)

Use of Memory

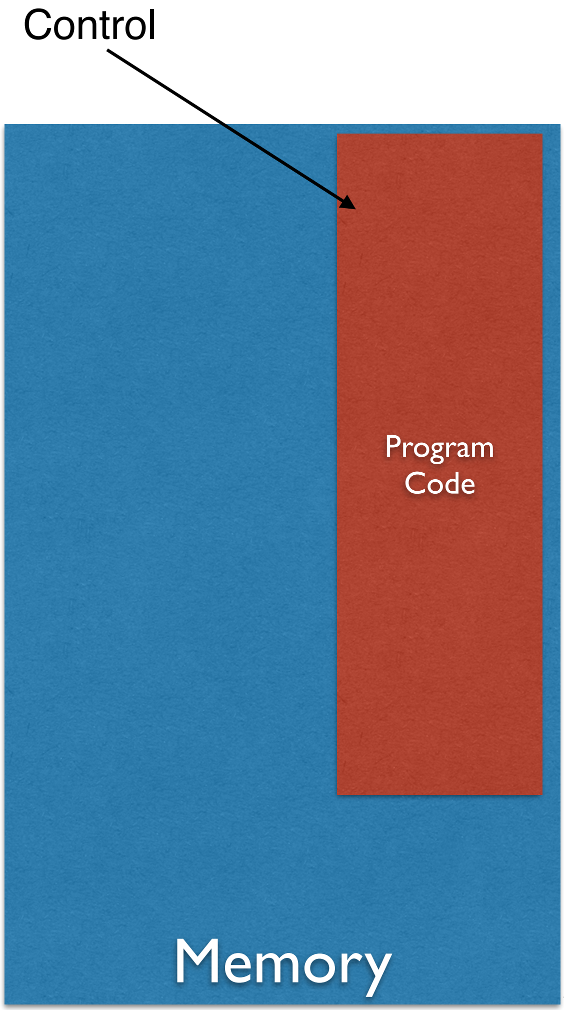


- Memory used for:
    - storing program code (executable)
    - storing program data
    - stack (to be discussed later)

Control

Hardware: Control

- Keeps track of what part of program is currently executing
- Each clock cycle, control points (in memory) to an instruction
- Instruction pointed to in that clock cycle performed (executed)
- Control moves to next instruction for next clock cycle
- Special instructions can move control (i.e., to do loops, branches, etc.)



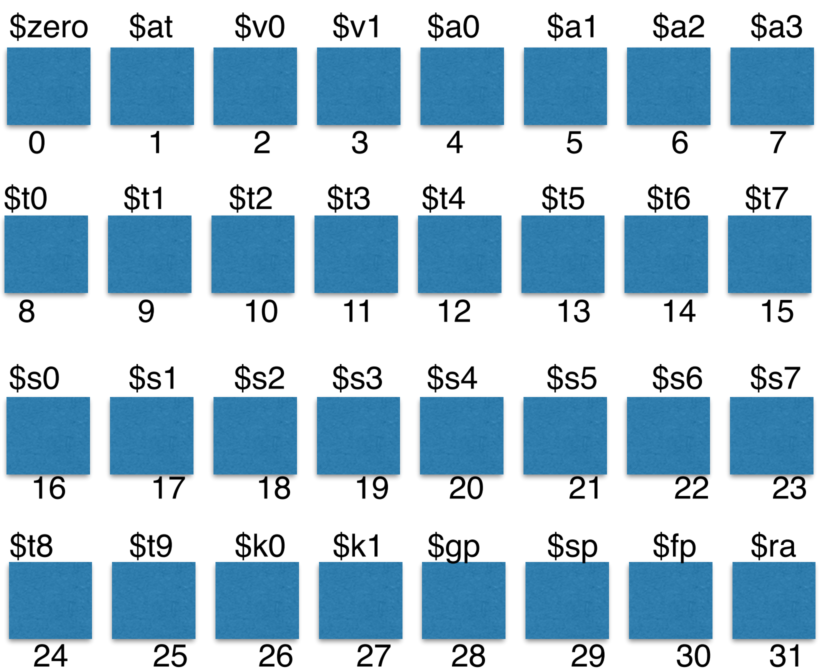
Registers and Register File

Hardware Component: Registers

- Collection of all registers called the Register File
- Each register can hold a word (wordsize # of bits) of data (over several clock cycles)
- Each register has a unique name (e.g., in MIPS: $s0, $s1, $t1, $sp, etc.)
- Computations can be performed on values in registers, e.g.,
  - Add the words in registers $s0 and $s1, store the resulting word in $s2
  - Logically AND the word in register $s0 with the constant 15, store result in $s0
- Values can be moved from memory to registers and back again, e.g.,
  - Load (copy-to-register) the word at memory location 4040 into register $s0
  - Store (copy-to-memory) the word in register $s1 into memory location 4000
  - Registers are special memories whose data can be processed
  - Kind of like (temporary) variables in a program

Register File & Registers

- Each register can hold a word (wordsize # of bits) of data (over several clock cycles)
- Each register has a unique name (e.g., in MIPS: $s0, $s1, $t1, $sp, etc.)



Register File & Registers

- Each register can hold a word (wordsize # of bits) of data (over several clock cycles)
- Each register has a unique name (e.g., in MIPS: $s0, $s1, $t1, $sp, etc.)

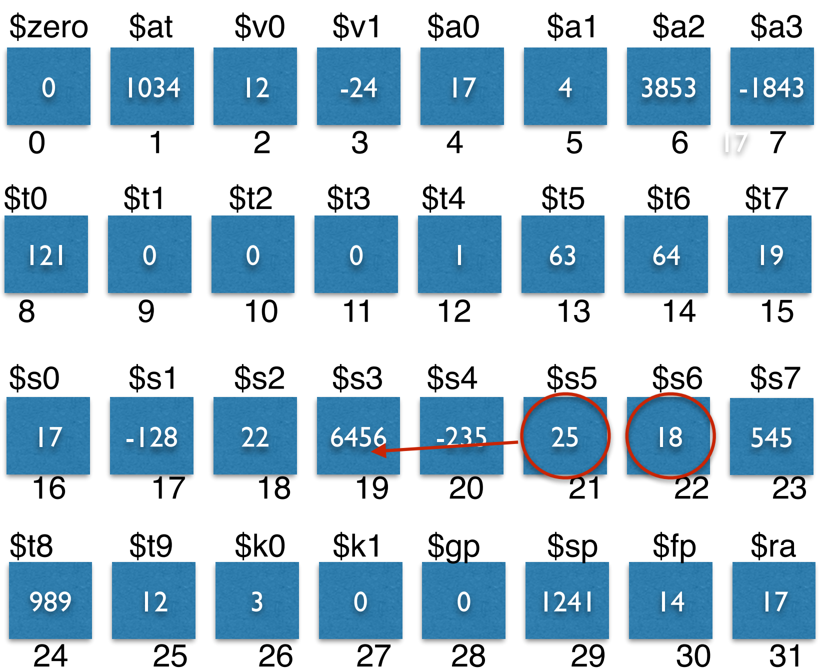
$zero	$at	$v0	$v1	$a0	$a1	$a2	$a3
0	1034	12	-24	17	4	3853	-1843
0	1	2	3	4	5	6	7
$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
121	0	0	0	1	63	64	19
8	9	10	11	12	13	14	15
$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
17	-128	22	6456	-235	25	18	545
16	17	18	19	20	21	22	23
$t8	$t9	$k0	$k1	$gp	$sp	$fp	$ra
989	12	3	0	0	1241	14	17
24	25	26	27	28	29	30	31

Each register stores a 32-bit quantity that can be changed during a clock cycle, or kept the same

Exception: $zero register always = 0

Register File & Registers

- Each register can hold a word (wordsize # of bits) of data (over several clock cycles)
- Each register has a unique name (e.g., in MIPS: $s0, $s1, $t1, $sp, etc.)



Each register stores a 32-bit quantity that can be changed during a clock cycle, or kept the same

Exception: $zero register always = 0

e.g., set $s3 = $s5 + $s6

Register File & Registers

- Each register can hold a word (wordsize # of bits) of data (over several clock cycles)
- Each register has a unique name (e.g., in MIPS: $s0, $s1, $t1, $sp, etc.)

$zero	$at	$v0	$v1	$a0	$a1	$a2	$a3
0	1034	12	-24	17	4	3853	-1843
0	1	2	3	4	5	6	7
$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
121	0	0	0	1	63	64	19
8	9	10	11	12	13	14	15
$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
17	-128	22	43	-235	25	18	545
16	17	18	19	20	21	22	23
$t8	$t9	$k0	$k1	$gp	$sp	$fp	$ra
989	12	3	0	0	1241	14	17
24	25	26	27	28	29	30	31

Each register stores a 32-bit quantity that can be changed during a clock cycle, or kept the same

Exception: $zero register always = 0

e.g., set $s3 = $s5 + $s6

next clock cycle: $s3 storing 43

Complete List (not important for course)

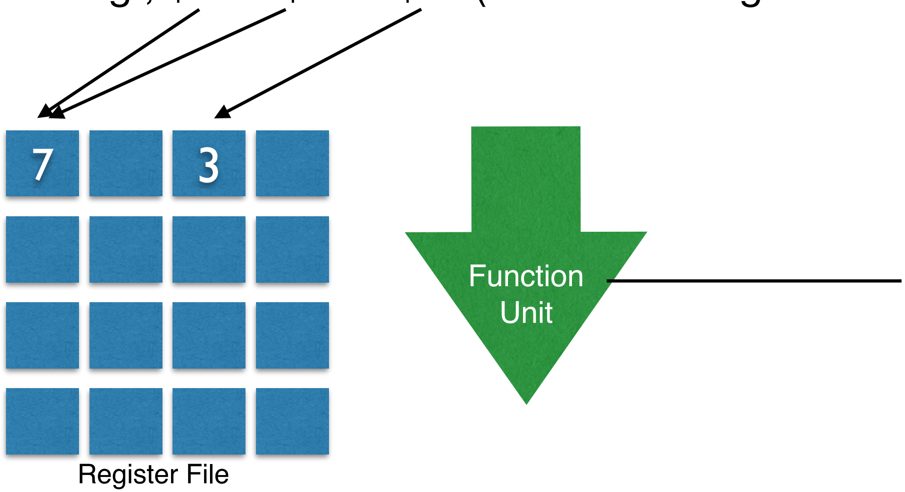
Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler (e.g., used by some multi-step pseudo-instructions)
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

FIGURE B.6.1 MIPS registers and usage convention.

Function Unit

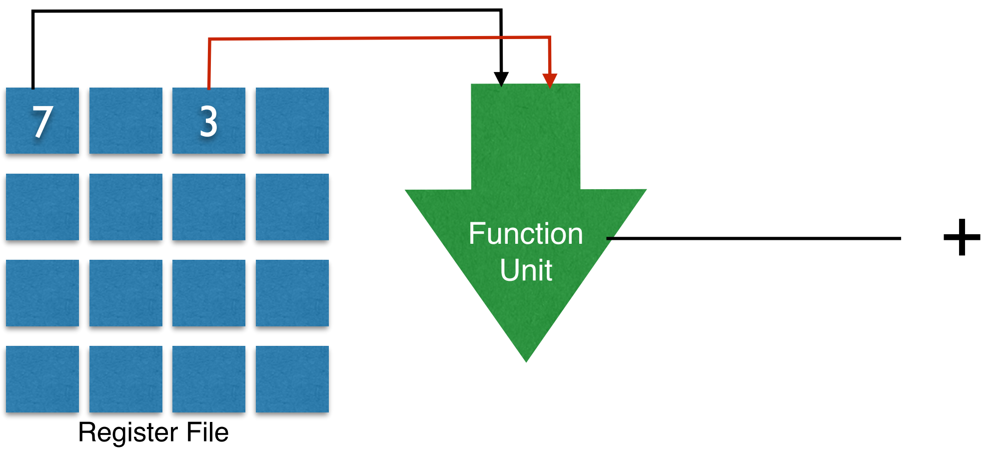
Hardware Component: Function Unit

- Not a holder of data (like memory and registers), but a processor of data
- Stateless (doesn't hold onto output)
  - Takes in Data (from registers or constant) and operation type (e.g., +, -, &, etc.)
  - Outputs result from applying operation to the input data
  - e.g., \$s0 = \$s0 + \$s2 (take vals in registers s0 and s2, add, store in s0)



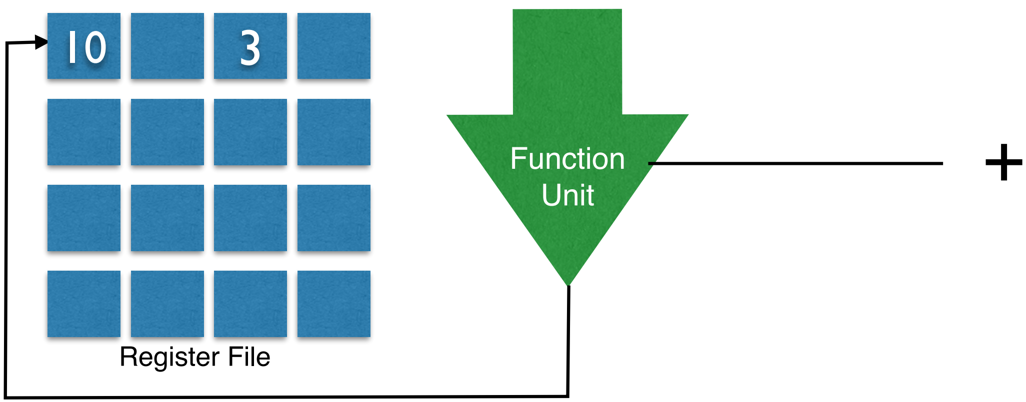
Function Unit

- Not a holder of data (like memory and registers), but a processor of data
- Stateless (doesn't hold onto output)
  - Takes in Data (from registers or constant) and operation type (e.g., +, -, &, etc.)
  - Outputs result from applying operation to the input data
  - e.g., $s0 = $s0 + $s2 (take vals in registers s0 and s2, add, store in s0)



Function Unit

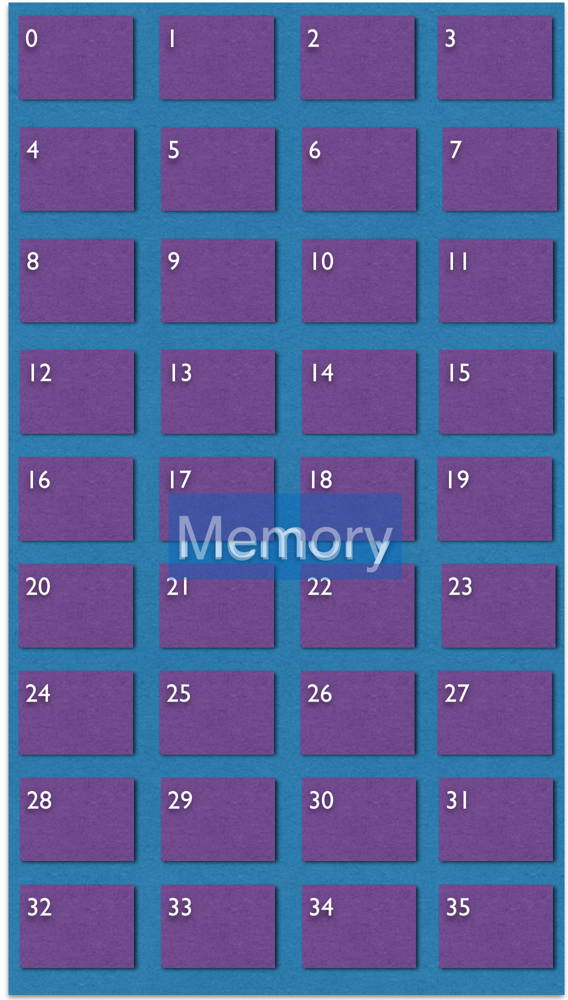
- Not a holder of data (like memory and registers), but a processor of data
- Stateless (doesn't hold onto output)
  - Takes in Data (from registers or constant) and operation type (e.g., +, -, &, etc.)
  - Outputs result from applying operation to the input data
  - e.g., $s0 = $s0 + $s2 (take vals in registers s0 and s2, add, store in s0)



Memory

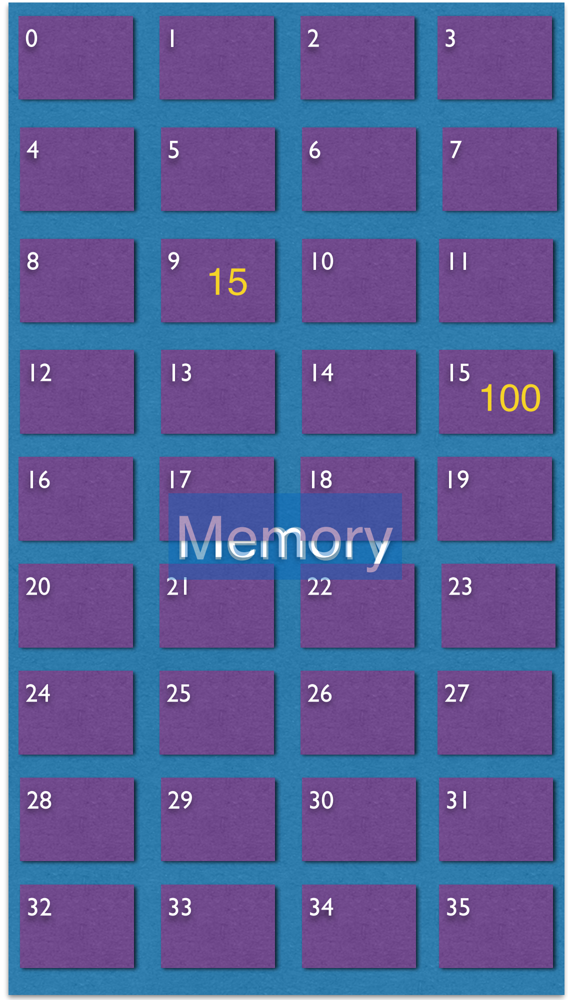
Hardware Component: Memory

- Just a big "Data storage locker"
- Can put data in memory and retrieve it later (over several clock cycles)
- Each memory location has a numerical address
  - In MIPS, each byte (8 bits) has an address
  - Usually access memory in MIPS one word at a time (32 bits, or 4 bytes)
  - e.g., (storing bytes)
    - store the value 15 in M[9]
    - store the value 100 in M[15]
    - read the value from M[9] (would return 15)



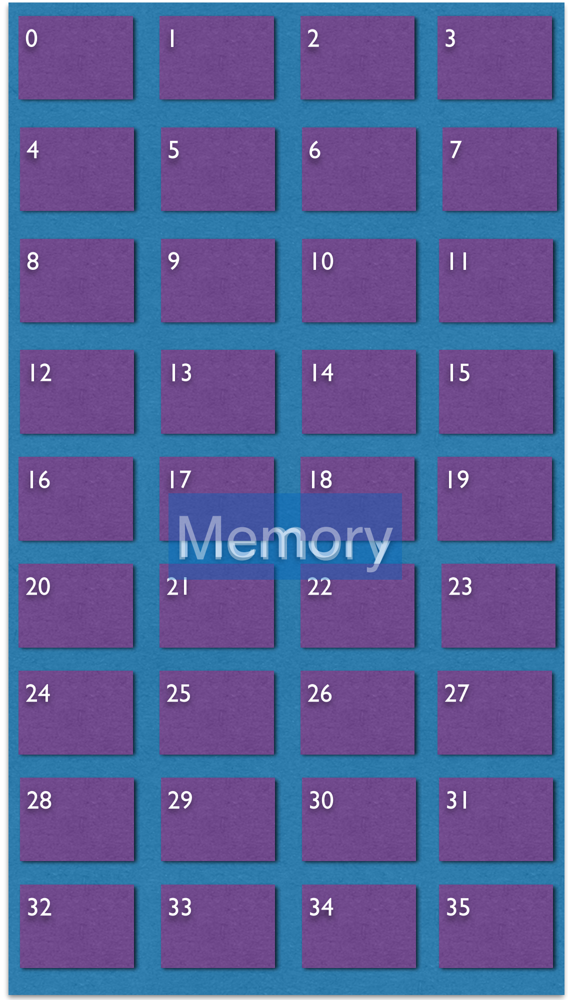
Hardware Component: Memory

- Just a big “Data storage locker”
- Can put data in memory and retrieve it later (over several clock cycles)
- Each memory location has a numerical address
  - In MIPS, each byte (8 bits) has an address
  - Usually access memory in MIPS one word at a time (32 bits, or 4 bytes)
  - e.g., (storing bytes)
    - store the value 15 in M[9]
    - store the value 100 in M[15]
    - read the value from M[9] (would return 15)



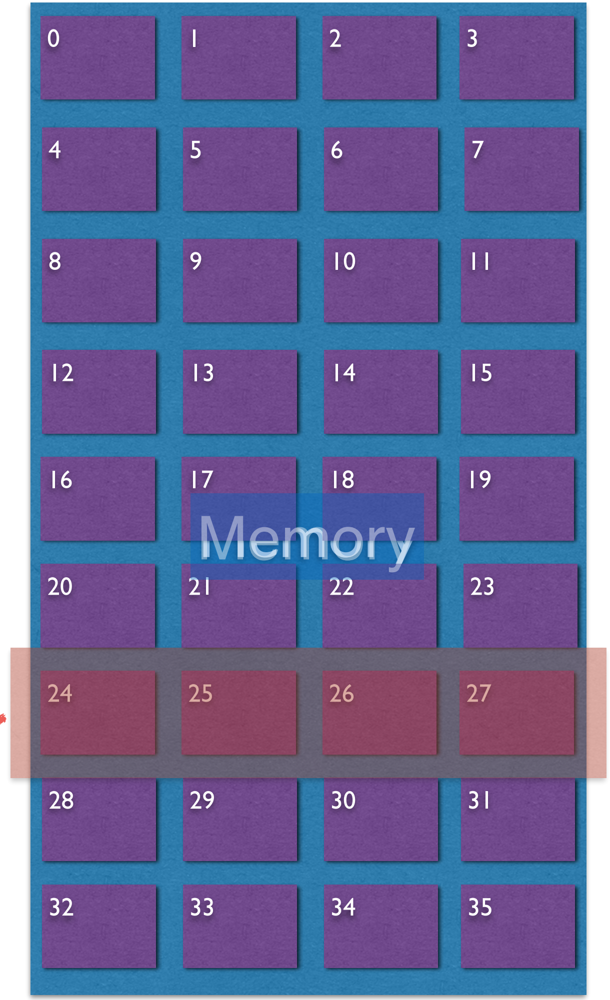
MIPS: 32 bit words and memory

- MIPS architecture has a 32-bit wordsize (4 bytes)
- Recall: each byte of memory has an address
- MIPS can access a word (4 bytes) at a time:
    - Specify the base address X (multiple of 4)
    - Data stored in addresses X, X+1, X+2, X+3 returned
- e.g., retrieve the word at address 24



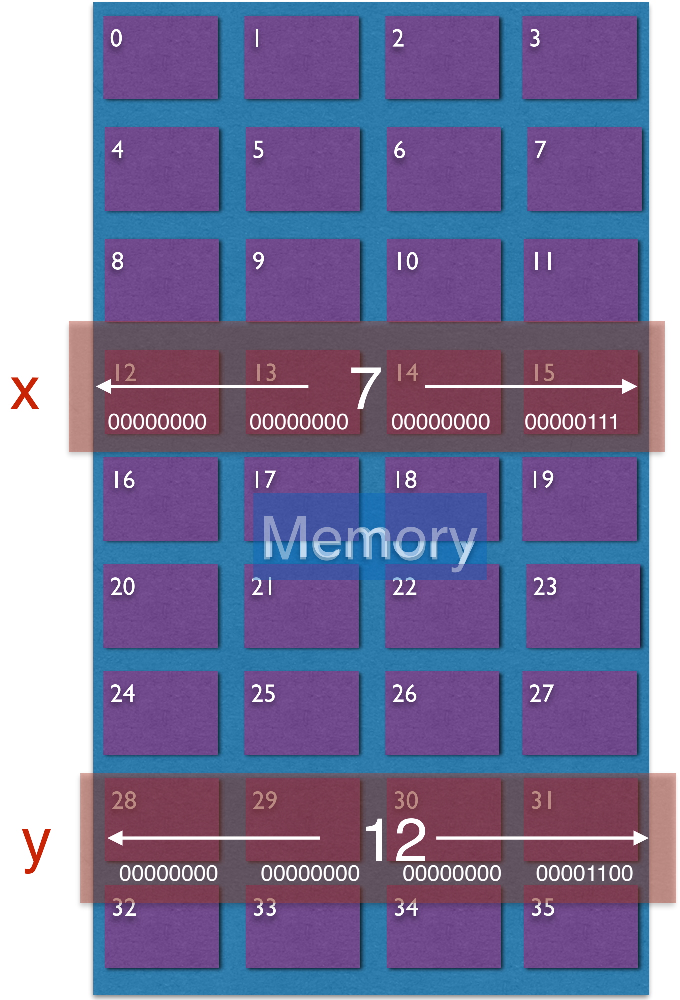
MIPS: 32 bit words and memory

- MIPS architecture has a 32-bit wordsize (4 bytes)
- Recall: each byte of memory has an address
- MIPS can access a word (4 bytes) at a time:
  - Specify the base address X (multiple of 4)
  - Data stored in addresses X, X+1, X+2, X+3 returned
- e.g., retrieve the word at address 24



Pointers (in C) and accessing memory in MIPS

- int x = 7;
  - x is a variable. It's value is stored somewhere in memory (or maybe a register)
- int *y = &x;
  - y is a pointer, pointing to (the address of) x. The value in y is the address where x's data is stored
  - y = 12 in this example
  - y is also a variable, and is stored somewhere in memory (or in a register), in this case, at address 28.



Memory v. Register in MIPS

Attribute	Memory	Register
Quantity	2^{32} bytes = 2^{30} words	32 registers in total
Functionality	Store to and Load from only	Load, Store, Math ops, etc.
Cost per word	Low	High
Speed to access	Slow	Fast

Instruction Set Arch
(ISA)

What is an ISA?

- An **Instruction Set Architecture**, or ISA, is an **interface** between the hardware and the software.

- An ISA consists of:

  - a set of operations (instructions)

  - data units (sized, addressing modes, etc.)

  - processor state (registers)

  - input and output control (memory operations)

  - execution model (program counter)

What is an ISA?

- An **Instruction Set Architecture**, or ISA, is an **interface** between the hardware and the software.

- An ISA consists of:

(for MIPS)

  - a set of operations (instructions) ← arithmetic, logical, conditional, branch, etc.
  - data units (sized, addressing modes, etc.) ← 32-bit data word
  - processor state (registers) ← 32, 32-bit registers
  - input and output control (memory operations) ← load and store
  - execution model (program counter) ← 32-bit program counter

Code

- A series of instructions that tells the function unit how to move and manipulate data

- High level language code:
  - e.g., C, C++, Java, Python, etc.
  - Portable: Divorced from architecture (as much as possible)
  - human-friendly
  - Compiled (e.g., C, C++, Java) or interpreted (Python) into low level language
- Low level language code:
  - e.g., MIPS, x86
  - Dependent on architecture (talking directly to the arch, not portable)
  - a lot less human friendly

Assembly Code vs. Machine Code

- A low-level language instruction has two forms: Assembly and Machine
  - **Assembly**: human-readable form, e.g., `add $t1, $s0, $s2`
    - says take values in registers s0 and s2, add them together, store result in register t1
  - **Machine**: a word (32 bits in MIPS) that is actually stored in memory that fully describes the instruction
    - e.g., add $t1, $s0, $s2 is 00000010 00110010 01000000 00100000 in binary or 02 32 40 20 in hex

- An **assembler** is software that converts a text file of assembly code (instructions) into a binary file of machine code
  - very straightforward (trivial) process: each instruction converts quite easily
  - One "smart" thing assembler does is permit labels for branches and jumps (discussed more later).

Assembly Program

- An Assembly program is a collection of instructions
- Each instruction might do several of the following:
  - Read from register file, from memory, or read in a constant
  - Perform an operation (e.g., +, -, &, |, comparison)
  - Write to register file or memory
  - "Jump" in the program control: decide which instruction should be next (e.g., like a GOTO)
- Complex operations are performed via sequences of instructions

Programming in MIPS

An example Program in MIPS: Factorial(n)

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

C 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 return
```

MIPS code

push

n items

pop

n items

base case

recursive call

An example Program in MIPS: Factorial(n)

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

C 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 return
```

MIPS code

Each line contains (at most) a single instruction

An example Program in MIPS: Factorial(n)

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

C code

Comments

```
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
```

MIPS code

A Program in MIPS

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

C code

Operations (what op the current instruction will perform during the current clock cycle)

```
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
```

MIPS code

A Program in MIPS

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

C code

Parameters that the instruction applies its operation to.

There are a few different types of parameters…

```
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
```

MIPS code

A Program in MIPS

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

C code

Registers

```
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
```

MIPS code

A Program in MIPS

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

C code

Constants

```
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
```

MIPS code

A Program in MIPS

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

C 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 return
```

MIPS code

Labels: used for control-oriented instructions (jumps and branches)

A Program in MIPS

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

C code

Instructions that perform arithmetic/ logic ops: store result of operation in a register

```
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
```

MIPS code

A Program in MIPS

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

C code

Instructions that transfer between register and main memory

```
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
```

MIPS code

A Program in MIPS

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

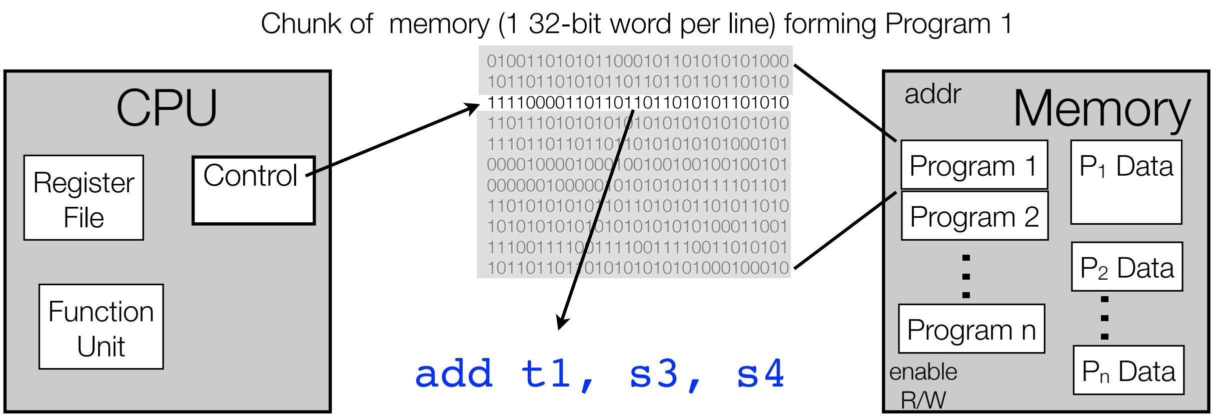
C code

Labels and "Jump/branch" Instructions that affect control (i.e., determine which instruction comes next)

```
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
```

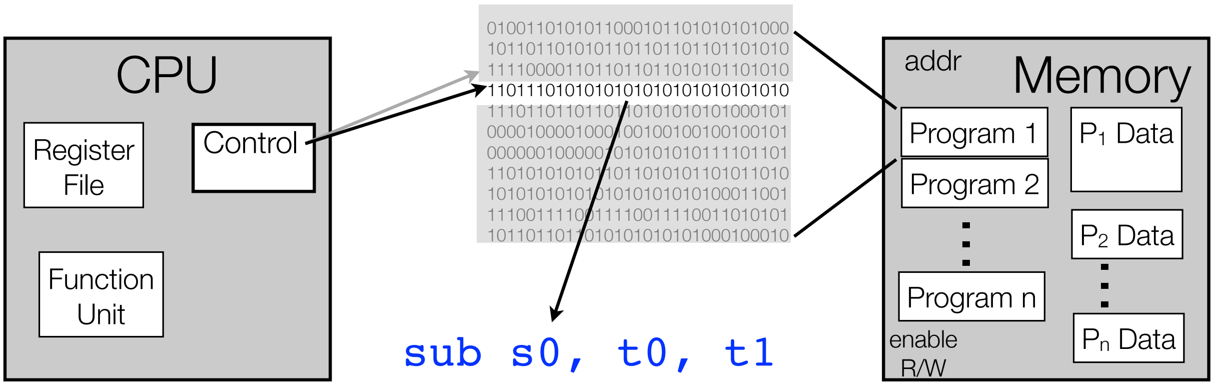
MIPS code

Simple View of ISA



- Control: points to the “current” instruction in memory and interprets it for execution
- Instruction then tells CPU what to do, e.g.,
  - “add values in registers s3 and s4 together, store result in register t1”
  - This example instruction
    - uses register file and function unit
    - does not read or write data from/to (main) memory

Simple View of ISA



- Normal control movement: Unless instruction specifies otherwise, after performing instruction, control moves to the next word in the memory (closest larger address) to get next instruction

Instruction Types

Instruction Types

- 4 basic types of instructions:
  - Affect State (values in registers, memory):
    - **Memory Access**: Move data to/from memory from/to registers
    - **Arithmetic/Logic**: Perform (via functional unit) computation on data in registers (and store result in a register)
  - Affect Control (which instruction is next to be processed):
    - **Jump / Jump Subroutine / Jump return**: direct control to a different part of the program (not next word in memory)
    - **Conditional branch**: test values in registers. If test returns true, move

Instructions of interest (Arithmetic/Logic)

rd, rs, rt are registers; const, shamt are constants

- add rd, rs, rt rd = rs+rt
- addi rt, rs, const rt = rs+const
- and rd, rs, rt rd = rs & rt
- or rd, rs, rt
- sll rd, rt, shamt rd = rt << shamt
- sub rd, rs, rt rd = rs - rt
- xor rd, rs, rt rd = rs ⊕ rt
- lui rt, const (load upr 16 bits of rt)
- slt rd, rs, rt (rd = 1 iff rs < rt, else 0)
- slti rt, rs, const (rt = 1 iff rs<const else 0)

- addu rd, rs, rt (unsigned)
- addiu rt, rs, const (unsigned)
- andi rt, rs, const
- ori rt, rs, const
- srl rd, rt, shamt rd = rt >> shamt
- xori rt, rs, const
- sltu rd, rs, rt (unsigned)
- sltui rt, rs, const (unsigned)

Immediate Operands

- Constant data encoded in an instruction

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

- No subtract immediate instruction, just use the negative constant

```
addi $s2, $s1, -1
```

MIPS Logical Operations

- Instructions for bitwise manipulation

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bit-by-bit AND	&	&	`and`, `andi`
Bit-by-bit OR	|	|	`or`, `ori`
Bit-by-bit NOT	~	~	`nor`

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

- Useful for inserting and extracting groups of bits in a word

The Constant (Register) Zero

- MIPS register 0 ($zero) is the constant 0
- $zero cannot be overwritten
- Useful for many operations, for example, a move between two registers

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

AND Operations

- example: `and $t0, $t1, $t2   # $t0 = $t1 & $t2`

- Useful for masking bits in a word (selecting some bits, clearing others to 0)

AND Operations

- example: `and $t0, $t1, $t2   # $t0 = $t1 & $t2`

- Useful for masking bits in a word (selecting some bits, clearing others to 0)

`$t1:`	`0000 0000 0000 0000 0000 1101 1100 0000`
`$t2:`	`0000 0000 0000 0000 0011 1100 0000 0000`
`$t0:`	`0000 0000 0000 0000 0000 1100 0000 0000`

OR Operations

- example: `or $t0, $t1, $t2  # $t0 = $t1 | $t2`

- Useful to include bits in a word (set some bits to 1, leaving others unchanged)

OR Operations

- example: `or $t0, $t1, $t2  # $t0 = $t1 | $t2`

- Useful to include bits in a word (set some bits to 1, leaving others unchanged)

`$t1:`	`0000 0000 0000 0000 0000 1101 1100 0000`
`$t2:`	`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

- MIPS has 3 operand NOR instruction, used to compute NOT
- example: `nor $t0, $t1, $zero  # $t0 = ~$t1`

$t1:	0000 0000 0000 0000 0000 1101 1100 0000
$t0:	1111 1111 1111 1111 1111 0010 0011 1111

Instructions of interest (control: jump and branch)

rd, rt, rsrc1, rsrc2 are registers
address is of form C[r] where C is const, r is register
target, label are labels

- beq rs, rt, label (rs==rt -> goto label)
- bgtz rs, label (rs > 0)
- j target (jump to label target)
- jalr rs, rd (jump and rd = return addr)
- bne rs, rt, label (rs!=rt -> goto label)
- bgez rs, label (rs >=0)
- jal target (jump with ability to return)
- jr ra (jump return to address in ra)

Instructions of interest (memory)

rd, rt, rsrc1, rsrc2 are registers
address is of form C[r] where C is const, r is register
target, label are labels

- lw rt, const(rs): rt = mem[(val in rs+const) (transfer mem->reg)
- sw rt, const(rs): mem[(val in rs+const)=rt (transfer reg->mem)

Examples translating C to MIPS

Arithmetic Example 1

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

C code

Arithmetic Example 1

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

C code

Assume following registers hold corresponding variables:

g : $s1
h : $s2
i : $s3
j : $s4
f : $s0

Arithmetic Example 1

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

C code

Assume following registers hold corresponding variables:

g : \$s1
h : \$s2
i : \$s3
j : \$s4
f : \$s0

```
add $t0, $s1, $s2   # temp $t0=g+h
add $t1, $s3, $s4   # temp $t1=i+j
sub $s0, $t0, $t1   # f = $t0-$t1
```

Compiled MIPS

In MIPS, instruction line can be followed by comments (following "#")

What’s in a Register?

```
int x;
unsigned int y;
float z;
int *p;
float *q;
```

What's in a Register?

```
int x;
unsigned int y;
float z;
int *p;
float *q;
```

$\left.\begin{array}{l}\texttt{int x;}\\\texttt{unsigned int y;}\\\texttt{float z;}\end{array}\right\}$ Data values

$\left.\begin{array}{l}\texttt{int *p;}\\\texttt{float *q;}\end{array}\right\}$ Pointers to memory

What's in a Register?

```
int x;
unsigned int y;
float z;
int *p;
float *q;
```

In MIPS, these are all 32-bit values

What's in a Register?

```
int x;
unsigned int y;
float z;
int *p;
float *q;
```

In MIPS, these are all 32-bit values

Each could be stored in a register or in a word of memory

Memory Operand Example 1

```
g = h + A[8];
```

C code

Memory Operand Example 1

```
g = h + A[8];
```

C code

Assume: Non-array variables stored in registers, arrays stored in main memory (but the location in memory of the array stored in a register)

Memory Operand Example 1

```
g = h + A[8];
```

C code

Assume: Non-array variables stored in registers, arrays stored in main memory (but the location in memory of the array stored in a register)

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)

Memory Operand Example 1

```
g = h + A[8];
```

C code

Assume: Non-array variables stored in registers, arrays stored in main memory (but the location in memory of the array stored in a register)

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)

Example: suppose array A[] stored starting at address 1000, where:
A[0] = 7
A[1] = -1
A[2] = 35
A[3] = 0

A[]	Mem Addr	stored
A[0] = 7		
A[1] = -1		
A[2] = 35		
A[3] = 0		

Memory Operand Example 1

```
g = h + A[8];
```

C code

Assume: Non-array variables stored in registers, arrays stored in main memory (but the location in memory of the array stored in a register)

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)

Example: suppose array A[] stored starting at address 1000, where:
A[0] = 7
A[1] = -1
A[2] = 35
A[3] = 0

A[]	Mem Addr	stored
A[0] = 7	1000-1003	00…00111
A[1] = -1	1004-1007	111…1111
A[2] = 35	1008-1011	00..00100011
A[3] = 0	1012-1015	00…00

Memory Operand Example 1

```
g = h + A[8];
```

C code

Assume: Non-array variables stored in registers, arrays stored in main memory (but the location in memory of the array stored in a register)

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)

\$s3 = 1000 (base address)
\$s3 + 0: address of A[0]
\$s3 + 4: address of A[1]
\$s3 + 8: address of A[2]
\$s3 + 4i: address of A[i]

A[]	Mem Addr	stored
A[0] = 7	1000-1003	00…00111
A[1] = -1	1004-1007	111…1111
A[2] = 35	1008-1011	00..00100011
A[3] = 0	1012-1015	00…00

Memory Operand Example 1

```
g = h + A[8];
```

C code

Assume: Non-array variables stored in registers, arrays stored in main memory (but the location in memory of the array stored in a register)

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)

- Recall memory operations: load (read from men) and store (write to mem)
  - MIPS Can only read from or write to memory
  - We only consider memory operations that transfer whole words (4 bytes)
  - Read: lw rt, const(rs): rt = mem[(val in rs+const) (transfer mem->reg)
  - Write: sw rt, const(rs): mem[(val in rs+const)=rt (transfer reg->mem)

Memory Operand Example 1

```
g = h + A[8];
```

C code

Assume: Non-array variables stored in registers, arrays stored in main memory (but the location in memory of the array stored in a register)

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

index = 8 requires offset of 32 (8 items x 4 bytes per word)

offset *base register*

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

Compiled MIPS

Memory Operand Example 2

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

C code

Memory Operand Example 2

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

C code

Assume h in $s2, base address of A in $s3

index = 8 requires offset of 32 (8 items x 4 bytes per word)
index = 12 requires offset of 48 (12 items x 4 bytes per word)

Memory Operand Example 2

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

C code

h in $s2, base address of A in $s3

index = 8 requires offset of 32 (8 items x 4 bytes per word)
index = 12 requires offset of 48 (12 items x 4 bytes per word)

```
lw $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw $t0, 48($s3)    # store word
```

Compiled MIPS

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- Instruction labeled with colon e.g. `L1: add $t0, $t1, $t2`
- Labels permit assembler to move code within memory: fixed values are determined (by the assembler) during compilation process
- `beq $rs, $rt, L1 # if (rs==rt) branch to instr labeled L1`
- `bne $rs, $rt, L1 # if (rs!=rt) branch to instr labeled L1`
- `j L1               # unconditional jump to instr labeled L1`

Examples with Conditionals/Branches

Compiling an If Statement

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

C code

- Assume f is in $s0, g is in $s1, h is in $s2, i is in $s3, j is in $s4
- The assembler calculates the addresses corresponding to the labels

Compiling an If Statement

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

C code

```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
Else:
  sub $s0, $s1, $s2
Exit:
```

Compiled MIPS

- Assume f is in $s0, g is in $s1, h is in $s2, i is in $s3, j is in $s4
- The assembler calculates the addresses corresponding to the labels

Compiling an If Statement

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

C code

```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
Else:
  sub $s0, $s1, $s2
Exit:
```

Labels

Compiled MIPS

- Assume f is in $s0, g is in $s1, h is in $s2, i is in $s3, j is in $s4
- The assembler calculates the addresses corresponding to the labels

Compiling an If Statement

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

C code

```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
Else:
  sub $s0, $s1, $s2
Exit:
```

Compiled MIPS

- Assume f is in $s0, g is in $s1, h is in $s2, i is in $s3, j is in $s4
- The assembler calculates the addresses corresponding to the labels

Compiling an If Statement

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

C code

Label naming up to coder

```
  bne $s3, $s4, Zpxyz
  add $s0, $s1, $s2
  j Yaya
Zpxyz:
  sub $s0, $s1, $s2
Yaya:
```

Compiled MIPS

- Assume f is in $s0, g is in $s1, h is in $s2, i is in $s3, j is in $s4
- The assembler calculates the addresses corresponding to the labels

Compiling an If Statement

Leading with bne best matches C code (why?)

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

If not equal then

C code

```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
Else:
  sub $s0, $s1, $s2
Exit:
```

Compiled MIPS

- Assume f is in $s0, g is in $s1, h is in $s2, i is in $s3, j is in $s4

- The assembler calculates the addresses corresponding to the labels when assembly code stored in memory (more on this later…)

Compiling a Loop Statement

```
while (save[i] == k)
     i += 1;
```

C code

- Assume i is in $s3, k is in $s4, address of save in $s5

Compiling a Loop Statement

```
while (save[i] == k)
    i += 1;
```

C code

```
Loop:
  sll $t1, $s3, 2
  add $t1, $t1, $s5
  lw $t0, 0($t1)
  bne $t0, $s4, Exit
  addi $s3, $s3, 1
  j Loop
Exit:
```

Compiled MIPS

- Assume i is in $s3, k is in $s4, address of save in $s5

Compiling a Loop Statement

```
while (save[i] == k)
    i += 1;
```

C code

```
Loop:
  sll $t1, $s3, 2
  add $t1, $t1, $s5
  lw $t0, 0($t1)
  bne $t0, $s4, Exit
  addi $s3, $s3, 1
  j Loop
Exit:
```

Compiled MIPS

- Assume i is in $s3, k is in $s4, address of save in $s5

Suppose save[0] stored at address 10,000. Then $s5 = 10,000

Suppose i = 5, then $s3 = 5, and save[i] = save[5] @ address 10,020

Compiling a Loop Statement

```
while (save[i] == k)
      i += 1;
```

C code

```
Loop:
  sll $t1, $s3, 2
  add $t1, $t1, $s5
  lw $t0, 0($t1)
  bne $t0, $s4, Exit
  addi $s3, $s3, 1
  j Loop
Exit:
```

$s3 = 5

$t1 = 20

$t1 = 10,020

put value in mem @ address 10,020 into $t0

Compiled MIPS

- Assume i is in $s3, k is in $s4, address of save in $s5

Suppose save[0] stored at address 10,000. Then $s5 = 10,000

Suppose i = 5, then $s3 = 5, and save[i] = save[5] @ address 10,020

More Conditional Operations

- **Set** result to 1 if a condition is true

- `slt rd, rs, rt` # (rs < rt) ? rd=1 : rd=0

- `slti rd, rs, constant` # (rs < constant) ? rd=1 : rd=0

- Use in combination with `beq` or `bne`

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

Branch Instruction Design

- Why not blt, bge, etc.?
- Hardware for <, >= etc. is slower than for = and !=
  - Combining with a branch involves more work per instruction, requiring a slower clock
  - All instructions would be "penalized" by slower clock because of this
- As beq and bne are the common case, this is a good compromise

Signed v. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- 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 thus $t0=1
sltu $t0, $s0, $s1 # unsigned: 4,294,967,295 > 1 thus $t0=0
```

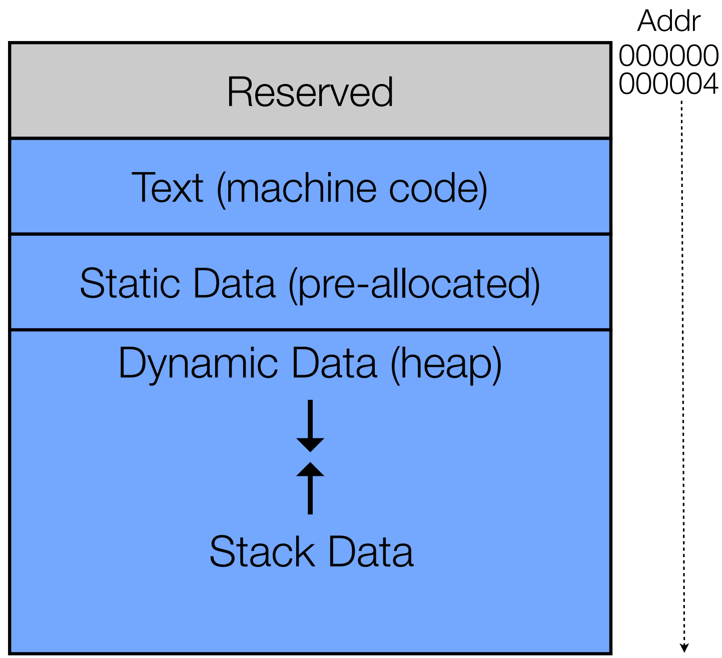
Using Memory:

Program Counter ($pc)

The Stack and stack pointer ($sp)

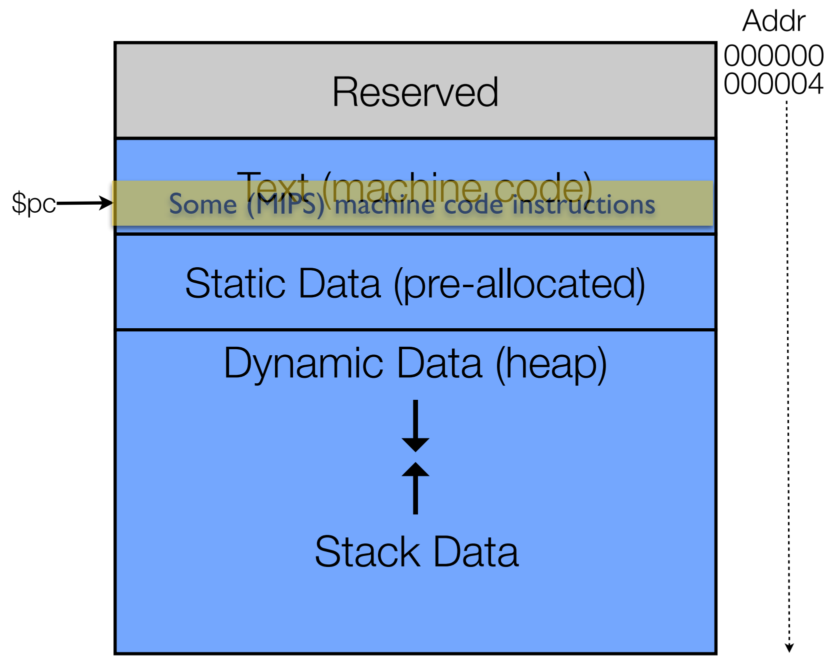
Memory Usage

- To be consistent with book, in this picture, higher addresses more toward top
- Memory has several uses
  - Reserved: space reserved for ?
  - Text/Code: where the program instructions are stored
  - Static data: pre-allocated memory
  - Dynamic data: data that can be allocated during runtime
  - Stack data: special stack structure useful for procedure calls



Memory Pointers: program counter ($pc)

- Program counter $pc "points" to a memory address
- $pc is not in the register file (special external register)
- The value $pc holds during a clock cycle is the memory address of the current instruction to be processed (by the microprocessor)



Compiling a Loop Statement

```
while (save[i] == k)
      i += 1;
```

C code

$pc
40,000

Mem Addr	
	`Loop:`
40,000	`sll $t1, $s3, 2`
40,004	`add $t1, $t1, $s5`
40,008	`lw $t0, 0($t1)`
40,012	`bne $t0, $s4, Exit`
40,016	`addi $s3, $s3, 1`
40,020	`j Loop`
	`Exit:`
40,024	`addi ….`

Compiling a Loop Statement

```
while (save[i] == k)
    i += 1;
```

C code

$pc
40,000

Mem Addr

	Loop:
40,000	sll $t1, $s3, 2
40,004	add $t1, $t1, $s5
40,008	lw $t0, 0($t1)
40,012	bne $t0, $s4, Exit
40,016	addi $s3, $s3, 1
40,020	j Loop
	Exit:
40,024	addi ….

Compiling a Loop Statement

```
while (save[i] == k)
    i += 1;
```

C code

$pc
40,004

Mem Addr

	Loop:
40,000	sll $t1, $s3, 2
40,004	add $t1, $t1, $s5
40,008	lw $t0, 0($t1)
40,012	bne $t0, $s4, Exit
40,016	addi $s3, $s3, 1
40,020	j Loop
	Exit:
40,024	addi ….

Compiling a Loop Statement

```
while (save[i] == k)
    i += 1;
```

C code

$pc
40,008

Mem Addr

```
        Loop:
40,000    sll $t1, $s3, 2
40,004    add $t1, $t1, $s5
40,008    lw $t0, 0($t1)
40,012    bne $t0, $s4, Exit
40,016    addi $s3, $s3, 1
40,020    j Loop
        Exit:
40,024    addi ….
```

Compiling a Loop Statement

```
while (save[i] == k)
    i += 1;
```

C code

$pc
40,012

Mem Addr	
	`Loop:`
40,000	`sll $t1, $s3, 2`
40,004	`add $t1, $t1, $s5`
40,008	`lw $t0, 0($t1)`
40,012	`bne $t0, $s4, Exit`
40,016	`addi $s3, $s3, 1`
40,020	`j Loop`
	`Exit:`
40,024	`addi ….`

Value of $pc for next clock cycle depends on outcome of bne instruction
Let's assume the condition holds true ($t0≠$s4) so code branches to Exit label…

Compiling a Loop Statement

```
while (save[i] == k)
     i += 1;
```

C code

$pc
40,024

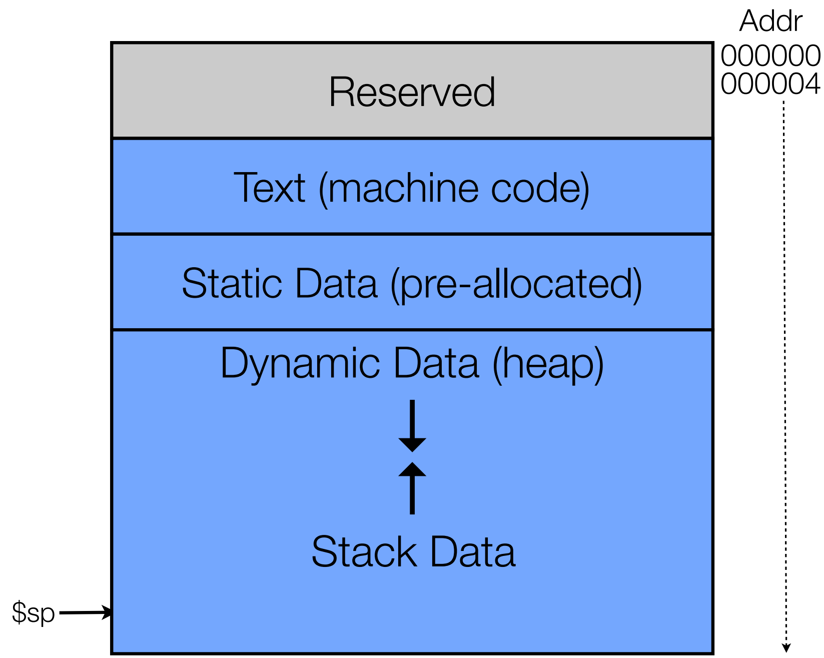
Mem Addr

```
        Loop:
40,000    sll $t1, $s3, 2
40,004    add $t1, $t1, $s5
40,008    lw $t0, 0($t1)
40,012    bne $t0, $s4, Exit
40,016    addi $s3, $s3, 1
40,020    j Loop
        Exit:
40,024    addi ….
```

Value of $pc for next clock cycle depends on outcome of bne instruction
Let’s assume the condition holds true ($t0≠$s4) so code branches to Exit label…

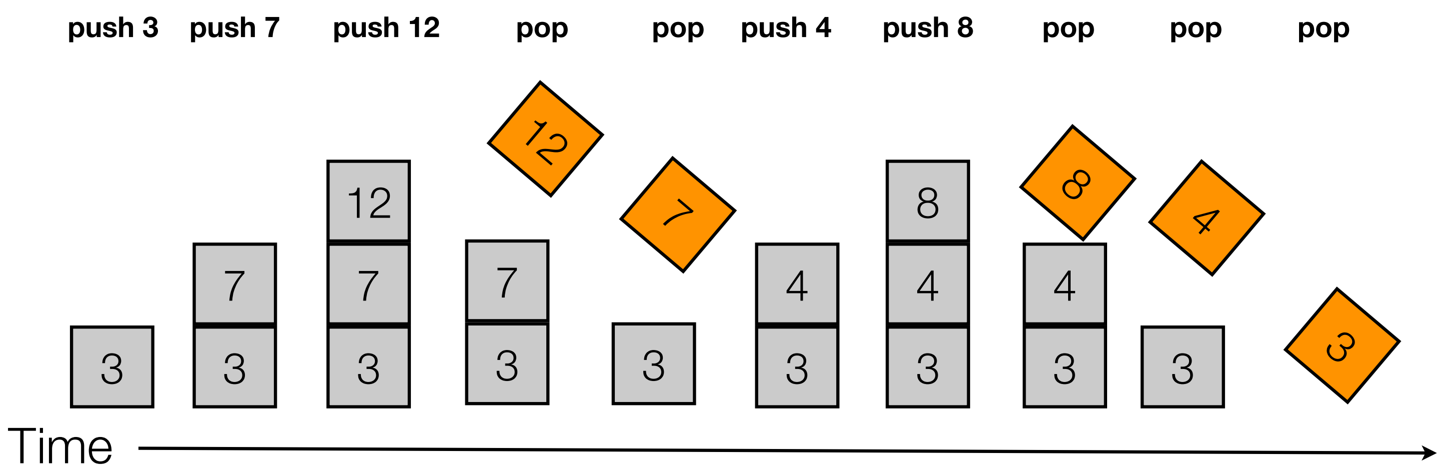
Memory Pointers

- Special registers (e.g., stack pointer $sp) "point" to memory address
- When "$sp points to address A" we mean
  - The value stored in $sp is A
  - We access the word in memory "pointed to" by $sp by sending the value in $sp to the address selector of memory
- Program counter ($pc) (indicates current instruction to execute) is also a memory pointer



Stack: Conceptual view

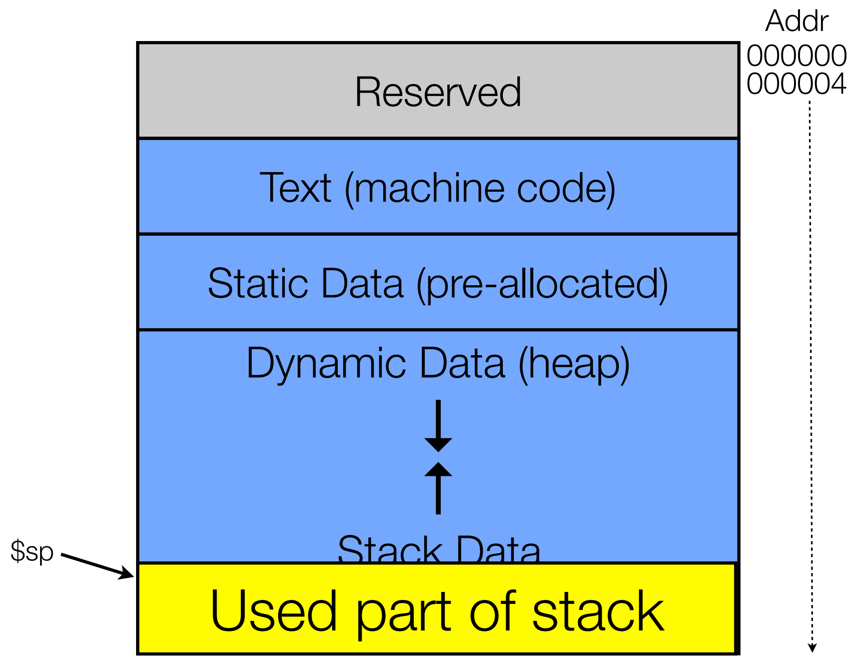
- Stack is a storage construct:
  - A new data item (word) is **push**ed onto the stack
  - Data items (words) can then be **pop**ped off the stack in the reverse order in which they were put on



- e.g., push 3, push 7, push 12, pop (returns 12), pop (returns 7), push 4, push 8, pop (returns 8), pop (returns 4), pop (returns 3)

Stack implementation in memory

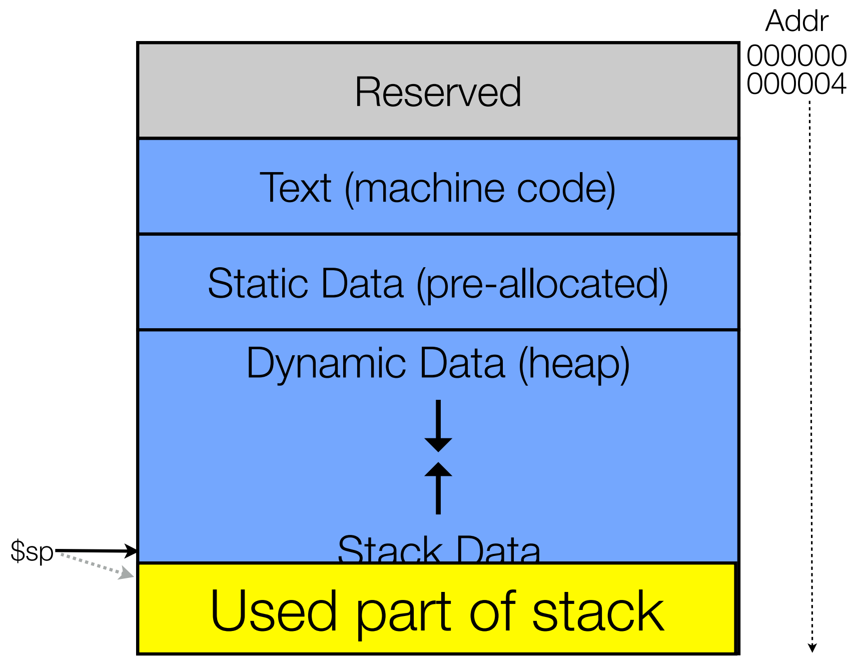
- Pushed word placed at highest available memory address
- Special stack pointer ($sp) register points to the lowest used address of the stack
- Push data D:
  - $sp -= 4;
  - store D in address pointed to by $sp



e.g., push 59

Stack implementation in memory

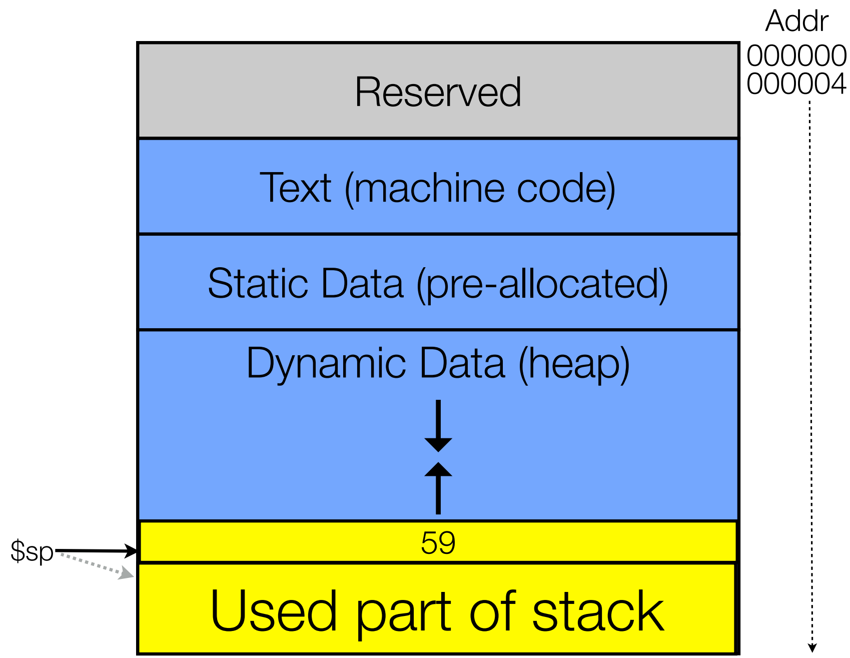
- Pushed word placed at highest available memory address
- Special stack pointer ($sp) register points to the lowest used address of the stack
- Push data D:
  - $sp -= 4;
  - store D in address pointed to by $sp



e.g., push 59

Stack implementation in memory

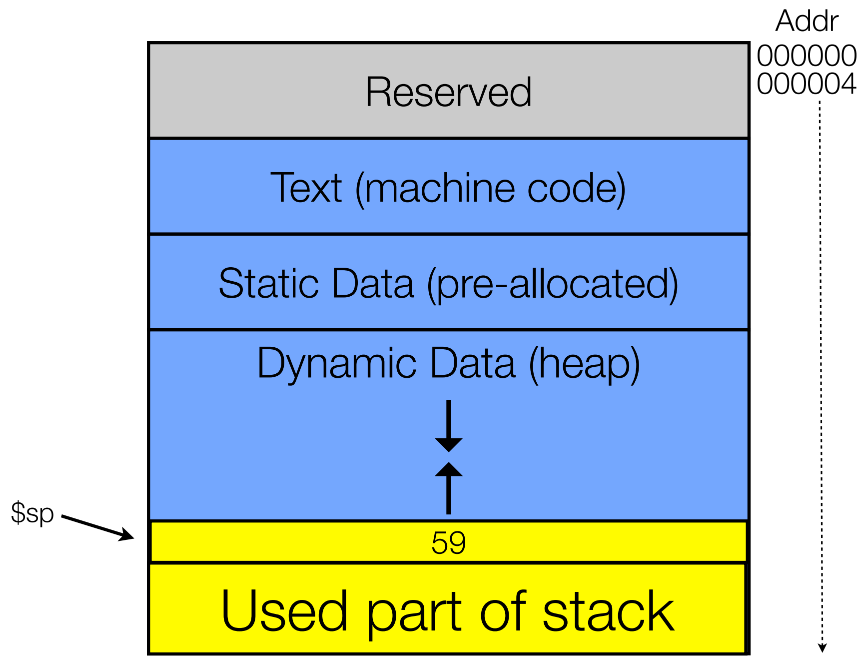
- Pushed word placed at highest available memory address
- Special stack pointer ($sp) register points to the lowest used address of the stack
- Push data D:
    - $sp -= 4;
    - store D in address pointed to by $sp



e.g., push 59

Stack implementation in memory

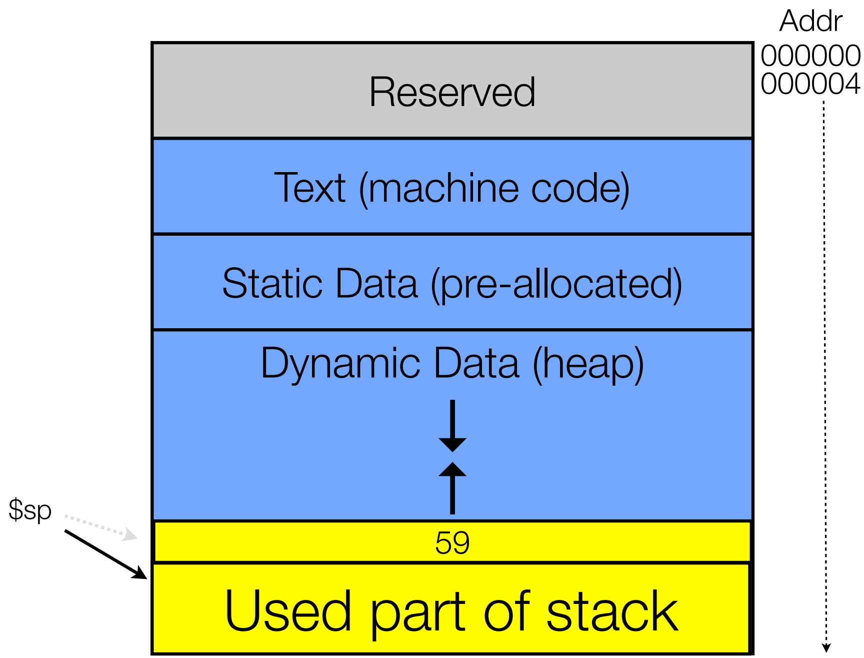
- Pushed word placed at highest available memory address
- Special stack pointer ($sp) register points to the lowest used address of the stack
- Push data D:
  - $sp -= 4;
  - store D in address pointed to by $sp



e.g., push 59

Stack implementation in memory

- Pop data D:
  - retrieve data word D in address pointed to by $sp
  - $sp += 4;
- Note: data "popped" not explicitly erased from memory
- But will be overwritten next time data is pushed onto stack



e.g., push 59

Procedure Calling

- Like function call in C program
  - control “jumps” to subroutine, and returns when complete
  - need to be careful about how “global” and “local” variables (registers) are used
- Steps required:
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call

Default Register Usage for procedures

- $a0-$a3: arguments (procedure can be passed 4 values through registers - passing more requires another approach, e.g., use of stack)
- $v0, $v1: result values: 2 return values permitted
- $t0-$t9: temporaries, can be overwritten by callee
- $s0-$s7: contents saved (must be restored by callee)
- $gp: global pointer for static data
- $sp: stack pointer
- $fp: frame pointer: useful as a fixed offset during a procedure to reference dynamic variables (the $sp might change values over time)
- $ra: return address
- Note: there is nothing special about these registers' design, only their implied use!!!
    - e.g., could store return value in $sp if calling and callee program both agreed to do this - just beware of messing up the stack for all other programs if not properly restored

Cross-call Data Preservation

- Nothing special about registers (from hardware perspective) - this is just their suggested usage!

Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.11 What is and what is not preserved across a procedure call. If the software relies on the frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

Procedure Calls
(and using the Stack)

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)
- When used together, jump to routine, and returns to subsequent instruction

Procedure Example

```
int leaf_example(int g,h,i,j) {
    int f;
    f = (g+h) - (i+j);
    return f+1;
}
```

C code

- Arguments g, h, i, j in $a0 - $a3
- f will (temporarily) go in $s0 (so will have to save previous contents of $s0 to stack)
- result (f+1) in $v0

Leaf Procedure Example

```
int leaf_example(int g,h,i,j) {
    int f;
    f = (g+h) - (i+j);
    return f+1;
}
```

C code

```
main(){
  int g;
  int f = 20;
  g = leaf_example(3,2,1,0);
  print("%d\n"g+f);
}
```

Suppose both main() and leaf_example store their (different) f variables in register $s0

- Arguments g, h, i, j in $a0 - $a3
- f will (temporarily) go in $s0 (so will have to save previous contents of $s0 to stack)
- result (f+1) in $v0

Leaf Procedure Example 2

```
int leaf_example(int g,h,i,j) {
    int f;
    f = (g+h) - (i+j);
    return f+1;
}
```

C code

Compiled MIPS	
`leaf_example:`	
`addi $sp, $sp, -4` `sw $s0, 0($sp)`	*save $s0 on stack*
`add $t0, $a0, $a1` `add $t1, $a2, $a3` `sub $s0, $t0, $t1`	*procedure body*
`addi $v0, $s0, 1`	*result*
`lw $s0, 0($sp)` `addi $sp, $sp, 4`	*restore $s0*
`jr $ra`	*return*

Compiled MIPS

Non-Leaf Procedures

- A non-leaf procedure is a procedure that calls another procedure
- For a nested call, the caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- After the call, the caller must restore these values from the stack

Non-Leaf Procedure Example

```c
int main() {
    int x = 5;
    int y = f2(x);
    printf(“y is %d\n”, y);
}

int f2(int a){
    int x = a+5;
    int y = f3(x);
    return y+x+a;
}

int f3(int a){
    return a+3;
}
```

C code

Assume any instance of:

- x stored in $s0
- y stored in $s1
- a stored in $a0 (input param)

Emulate c code as close as possible

Non-Leaf Procedure Example

```
int main() {
    int x = 5;
    int y = f2(x);
    printf(“y is %d\n”, y);
}

int f2(int a){
    int x = a+5;
    int y = f3(x);
    return y+x+a;
}

int f3(int a){
    return a+3;
}
```

C code

Assume any instance of:
- x stored in $s0
- y stored in $s1
- a stored in $a0 (input param)

Emulate c code as close as possible

```
main:
     addi $s0, $zero, 5
     mov $a0, $s0
     jal f2
     mov $s1, $v0
    <code to print and exit>
```

`f2: addi $sp, $sp, -8` `sw $s0, 4($sp)` `sw $s1, 0($sp)`	Push calling routine $s0 and $s1 onto the stack
`addi $s0, $a0, 5`	x = a+5
`addi $sp, $sp, -8` `sw $a0, 4($sp)` `sw $ra, 0($sp)`	Prepare for proc. call. Save f2’s a and $ra on stack
`mov $a0, $s0`	set param ($a0) to x val
`jal f3` `lw $ra, 0($sp)` `lw $a0, 4($sp)` `addi $sp, $sp, 8`	Recover f2’s a and $ra on stack
`mov $s1, $v0`	y = (result of f3(x))
`add $v0, $s1, Ss0` `add $v0, $v0, $a0`	compute val to return (y+x+a)
`lw $s0, 4($sp)` `lw $s1, 0($sp)` `addi $sp, $sp, 8` `jr $ra`	restore calling routine’s $s0 and $s1 from stack & return
`f3: addi $v0, $a0, 3` `jr $ra`	

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
1020→5	-138	19	62

$ra
2460

$pc
40000

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Push calling routine $s0 and $s1 onto the stack

x = a+5

Prepare for proc. call. Save f2's a and $ra on stack

set param ($a0) to x val

Recover f2's a and $ra on stack

y = (result of f3(x))

compute val to return (y+x+a)

restore calling routine's $s0 and $s1 from stack & return

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	289342
80140	4
80144	17
80148	12
80152	16

$sp
80144

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
5	-138	19→5	62

$ra
2460

$pc
40004

$sp
80144

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	289342
80140	4
80144	17
80148	12
80152	16

```
      main:
40000     addi $s0, $zero, 5
40004     mov $a0, $s0
40008     jal f2
40012     mov $s1, $v0
40xxx     <code to print and exit>
40100 f2: addi $sp, $sp, -8
40104     sw $s0, 4($sp)
40108     sw $s1, 0($sp)
40112     addi $s0, $a0, 5
40116     addi $sp, $sp, -8
40120     sw $a0, 4($sp)
40124     sw $ra, 0($sp)
40128     mov $a0, $s0
40132     jal f3
40136     lw $ra, 0($sp)
40140     lw $a0, 4($sp)
40144     addi $sp, $sp, 8
40148     mov $s1, $v0
40152     add $v0, $s1, Ss0
40156     add $v0, $v0, $a0
40160     lw $s0, 4($sp)
40164     lw $s1, 0($sp)
40168     addi $sp, $sp, 8
40172     jr $ra
40176 f3: addi $v0, $a0, 3
40180     jr $ra
```

Push calling routine $s0 and $s1 onto the stack

x = a+5

Prepare for proc. call. Save f2's a and $ra on stack

set param ($a0) to x val

Recover f2's a and $ra on stack

y = (result of f3(x))

compute val to return (y+x+a)

restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
5	-138	5	62

$ra
2460→ 40012

$pc
40008→ 40100

$sp
80144

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal  f2
40012      mov $s1, $v0
40xxx     <code to print and exit>
40100 f2:  addi $sp, $sp, -8        # Push calling routine $s0
40104      sw $s0, 4($sp)           # and $s1 onto the stack
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5         # x = a+5
40116      addi $sp, $sp, -8        # Prepare for proc. call. Save
40120      sw $a0, 4($sp)           # f2's a and $ra on stack
40124      sw $ra, 0($sp)
40128      mov $a0, $s0             # set param ($a0) to x val
40132      jal f3
40136      lw $ra, 0($sp)           # Recover f2's a and $ra on
40140      lw $a0, 4($sp)           # stack
40144      addi $sp, $sp, 8
40148      mov $s1, $v0             # y = (result of f3(x))
40152      add $v0, $s1, Ss0        # compute val to
40156      add $v0, $v0, $a0        # return (y+x+a)
40160      lw $s0, 4($sp)           # restore calling routine's
40164      lw $s1, 0($sp)           # $s0 and $s1 from stack &
40168      addi $sp, $sp, 8         # return
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	289342
80140	4
80144	17
80148	12
80152	16

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
5	-138	5	62

$ra
40012

$pc
40100

```
       main:
40000       addi $s0, $zero, 5
40004       mov $a0, $s0
40008       jal f2
40012       mov $s1, $v0
40xxx       <code to print and exit>
40100  f2:  addi $sp, $sp, -8
40104       sw $s0, 4($sp)
40108       sw $s1, 0($sp)
40112       addi $s0, $a0, 5
40116       addi $sp, $sp, -8
40120       sw $a0, 4($sp)
40124       sw $ra, 0($sp)
40128       mov $a0, $s0
40132       jal f3
40136       lw $ra, 0($sp)
40140       lw $a0, 4($sp)
40144       addi $sp, $sp, 8
40148       mov $s1, $v0
40152       add $v0, $s1, Ss0
40156       add $v0, $v0, $a0
40160       lw $s0, 4($sp)
40164       lw $s1, 0($sp)
40168       addi $sp, $sp, 8
40172       jr $ra
40176  f3:  addi $v0, $a0, 3
40180       jr $ra
```

Push calling routine $s0 and $s1 onto the stack

x = a+5

Prepare for proc. call. Save f2's a and $ra on stack

set param ($a0) to x val

Recover f2's a and $ra on stack

y = (result of f3(x))

compute val to return (y+x+a)

restore calling routine's $s0 and $s1 from stack & return

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	289342
80140	4
80144	17
80148	12
80152	16

$sp
80144
80136

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0	$ra	$pc	$sp
5	-138	5	62	40012	40104	80136

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	289342
80140	4→5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

- Push calling routine $s0 and $s1 onto the stack (40100–40108)
- x = a+5 (40112)
- Prepare for proc. call. Save f2's a and $ra on stack (40116–40124)
- set param ($a0) to x val (40128)
- Recover f2's a and $ra on stack (40136–40144)
- y = (result of f3(x)) (40148)
- compute val to return (y+x+a) (40152–40156)
- restore calling routine's $s0 and $s1 from stack & return (40160–40172)

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0	$ra	$pc	$sp
5	-138	5	62	40012	40108	80136

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	289342→-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

- 40100–40108: Push calling routine $s0 and $s1 onto the stack
- 40112: x = a+5
- 40116–40124: Prepare for proc. call. Save f2's a and $ra on stack
- 40128: set param ($a0) to x val
- 40136–40144: Recover f2's a and $ra on stack
- 40148: y = (result of f3(x))
- 40152–40156: compute val to return (y+x+a)
- 40160–40172: restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0	$ra	$pc	$sp
5→10	-138	5	62	40012	40112	80136

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Push calling routine $s0 and $s1 onto the stack (40100–40108)

x = a+5 (40112)

Prepare for proc. call. Save f2's a and $ra on stack (40116–40124)

set param ($a0) to x val (40128)

Recover f2's a and $ra on stack (40136–40144)

y = (result of f3(x)) (40148)

compute val to return (y+x+a) (40152–40156)

restore calling routine's $s0 and $s1 from stack & return (40160–40172)

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	-138
80140	5
80144	17
80148	12
80152	16

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	5	62

$ra
40012

$pc
40116

```
      main:
40000     addi $s0, $zero, 5
40004     mov $a0, $s0
40008     jal f2
40012     mov $s1, $v0
40xxx     <code to print and exit>
40100 f2: addi $sp, $sp, -8
40104     sw $s0, 4($sp)
40108     sw $s1, 0($sp)
40112     addi $s0, $a0, 5
40116     addi $sp, $sp, -8
40120     sw $a0, 4($sp)
40124     sw $ra, 0($sp)
40128     mov $a0, $s0
40132     jal f3
40136     lw $ra, 0($sp)
40140     lw $a0, 4($sp)
40144     addi $sp, $sp, 8
40148     mov $s1, $v0
40152     add $v0, $s1, Ss0
40156     add $v0, $v0, $a0
40160     lw $s0, 4($sp)
40164     lw $s1, 0($sp)
40168     addi $sp, $sp, 8
40172     jr $ra
40176 f3: addi $v0, $a0, 3
40180     jr $ra
```

- Push calling routine $s0 and $s1 onto the stack
- x = a+5
- Prepare for proc. call. Save f2's a and $ra on stack
- set param ($a0) to x val
- Recover f2's a and $ra on stack
- y = (result of f3(x))
- compute val to return (y+x+a)
- restore calling routine's $s0 and $s1 from stack & return

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435
80136	-138
80140	5
80144	17
80148	12
80152	16

$sp
80136
80128

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	5	62

$ra	$pc	$sp
40012	40120	80128

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

- Push calling routine $s0 and $s1 onto the stack
- x = a+5
- Prepare for proc. call. Save f2's a and $ra on stack
- set param ($a0) to x val
- Recover f2's a and $ra on stack
- y = (result of f3(x))
- compute val to return (y+x+a)
- restore calling routine's $s0 and $s1 from stack & return

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985
80132	2435→5
80136	-138
80140	5
80144	17
80148	12
80152	16

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0	$ra	$pc	$sp
10	-138	5	62	40012	40124	80128

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	-243985→40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000     addi $s0, $zero, 5
40004     mov $a0, $s0
40008     jal f2
40012     mov $s1, $v0
40xxx     <code to print and exit>
40100 f2: addi $sp, $sp, -8
40104     sw $s0, 4($sp)
40108     sw $s1, 0($sp)
40112     addi $s0, $a0, 5
40116     addi $sp, $sp, -8
40120     sw $a0, 4($sp)
40124     sw $ra, 0($sp)
40128     mov $a0, $s0
40132     jal f3
40136     lw $ra, 0($sp)
40140     lw $a0, 4($sp)
40144     addi $sp, $sp, 8
40148     mov $s1, $v0
40152     add $v0, $s1, Ss0
40156     add $v0, $v0, $a0
40160     lw $s0, 4($sp)
40164     lw $s1, 0($sp)
40168     addi $sp, $sp, 8
40172     jr $ra
40176 f3: addi $v0, $a0, 3
40180     jr $ra
```

Push calling routine $s0 and $s1 onto the stack (40100–40108)

x = a+5 (40112)

Prepare for proc. call. Save f2's a and $ra on stack (40116–40124)

set param ($a0) to x val (40128)

Recover f2's a and $ra on stack (40136–40144)

y = (result of f3(x)) (40148)

compute val to return (y+x+a) (40152–40156)

restore calling routine's $s0 and $s1 from stack & return (40160–40172)

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	5→10	62

$ra
40012

$pc
40128

$sp
80128

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Push calling routine $s0 and $s1 onto the stack

x = a+5

Prepare for proc. call. Save f2's a and $ra on stack

set param ($a0) to x val

Recover f2's a and $ra on stack

y = (result of f3(x))

compute val to return (y+x+a)

restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	10	62

$ra
40012
40136

$pc
40132
40176

$sp
80128

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8        Push calling routine $s0
40104      sw $s0, 4($sp)           and $s1 onto the stack
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5         x = a+5
40116      addi $sp, $sp, -8        Prepare for proc. call. Save
40120      sw $a0, 4($sp)           f2's a and $ra on stack
40124      sw $ra, 0($sp)
40128      mov $a0, $s0             set param ($a0) to x val
40132      jal f3
40136      lw $ra, 0($sp)           Recover f2's a and $ra on
40140      lw $a0, 4($sp)           stack
40144      addi $sp, $sp, 8
                                    y = (result of f3(x))
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0        compute val to
40156      add $v0, $v0, $a0        return (y+x+a)
40160      lw $s0, 4($sp)           restore calling routine's
40164      lw $s1, 0($sp)           $s0 and $s1 from stack &
40168      addi $sp, $sp, 8         return
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	10	62→13

$ra
40136

$pc
40176

$sp
80128

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000     addi $s0, $zero, 5
40004     mov $a0, $s0
40008     jal f2
40012     mov $s1, $v0
40xxx     <code to print and exit>
40100 f2: addi $sp, $sp, -8
40104     sw $s0, 4($sp)
40108     sw $s1, 0($sp)
40112     addi $s0, $a0, 5
40116     addi $sp, $sp, -8
40120     sw $a0, 4($sp)
40124     sw $ra, 0($sp)
40128     mov $a0, $s0
40132     jal f3
40136     lw $ra, 0($sp)
40140     lw $a0, 4($sp)
40144     addi $sp, $sp, 8
40148     mov $s1, $v0
40152     add $v0, $s1, Ss0
40156     add $v0, $v0, $a0
40160     lw $s0, 4($sp)
40164     lw $s1, 0($sp)
40168     addi $sp, $sp, 8
40172     jr $ra
40176 f3: addi $v0, $a0, 3
40180     jr $ra
```

Push calling routine $s0 and $s1 onto the stack

x = a+5

Prepare for proc. call. Save f2's a and $ra on stack

set param ($a0) to x val

Recover f2's a and $ra on stack

y = (result of f3(x))

compute val to return (y+x+a)

restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	10	13

$ra
40136

$pc
40180
40136

$sp
80128

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Push calling routine $s0 and $s1 onto the stack

x = a+5

Prepare for proc. call. Save f2's a and $ra on stack

set param ($a0) to x val

Recover f2's a and $ra on stack

y = (result of f3(x))

compute val to return (y+x+a)

restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	10	13

$ra: 40136 (previously 40012)

$pc: 40136

$sp: 80128

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000     addi $s0, $zero, 5
40004     mov $a0, $s0
40008     jal f2
40012     mov $s1, $v0
40xxx     <code to print and exit>
40100 f2: addi $sp, $sp, -8        # Push calling routine $s0 and $s1 onto the stack
40104     sw $s0, 4($sp)
40108     sw $s1, 0($sp)
40112     addi $s0, $a0, 5         # x = a+5
40116     addi $sp, $sp, -8        # Prepare for proc. call. Save f2's a and $ra on stack
40120     sw $a0, 4($sp)
40124     sw $ra, 0($sp)
40128     mov $a0, $s0             # set param ($a0) to x val
40132     jal f3
40136     lw $ra, 0($sp)           # Recover f2's a and $ra on stack
40140     lw $a0, 4($sp)
40144     addi $sp, $sp, 8
40148     mov $s1, $v0             # y = (result of f3(x))
40152     add $v0, $s1, Ss0        # compute val to return (y+x+a)
40156     add $v0, $v0, $a0
40160     lw $s0, 4($sp)           # restore calling routine's $s0 and $s1 from stack & return
40164     lw $s1, 0($sp)
40168     addi $sp, $sp, 8
40172     jr $ra
40176 f3: addi $v0, $a0, 3
40180     jr $ra
```

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	10→5	13

$ra
40012

$pc
40140

$sp
80128

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Instructions	Comment
40100–40108	Push calling routine $s0 and $s1 onto the stack
40112	x = a+5
40116–40124	Prepare for proc. call. Save f2's a and $ra on stack
40128	set param ($a0) to x val
40136–40140	Recover f2's a and $ra on stack
40144	y = (result of f3(x))
40148–40156	compute val to return (y+x+a)
40160–40172	restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138	5	13

$ra
40012

$pc
40144

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

$sp
80128
80136

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Push calling routine $s0 and $s1 onto the stack

x = a+5

Prepare for proc. call. Save f2's a and $ra on stack

set param ($a0) to x val

Recover f2's a and $ra on stack

y = (result of f3(x))

compute val to return (y+x+a)

restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10	-138→13	5	13

$ra
40012

$pc
40148

$sp
80136

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8        Push calling routine $s0
40104      sw $s0, 4($sp)           and $s1 onto the stack
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5         x = a+5
40116      addi $sp, $sp, -8        Prepare for proc. call.  Save
40120      sw $a0, 4($sp)           f2's a and $ra on stack
40124      sw $ra, 0($sp)
40128      mov $a0, $s0             set param ($a0) to x val
40132      jal f3
40136      lw $ra, 0($sp)           Recover f2's a and $ra on
40140      lw $a0, 4($sp)           stack
40144      addi $sp, $sp, 8
40148      mov $s1, $v0             y = (result of f3(x))
40152      add $v0, $s1, Ss0        compute val to
40156      add $v0, $v0, $a0        return (y+x+a)
40160      lw $s0, 4($sp)           restore calling routine's
40164      lw $s1, 0($sp)           $s0 and $s1 from stack &
40168      addi $sp, $sp, 8         return
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0	$ra	$pc	$sp
10	13	5	11→23	40012	40152	80136

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8         # Push calling routine $s0 and $s1 onto the stack
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5          # x = a+5
40116      addi $sp, $sp, -8         # Prepare for proc. call. Save f2's a and $ra on stack
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0              # set param ($a0) to x val
40132      jal f3
40136      lw $ra, 0($sp)            # Recover f2's a and $ra on stack
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0              # y = (result of f3(x))
40152      add $v0, $s1, Ss0         # compute val to return (y+x+a)
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)            # restore calling routine's $s0 and $s1 from stack & return
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0	$ra	$pc	$sp
10	13	5	23→28	40012	40156	80136

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

- Push calling routine $s0 and $s1 onto the stack (40100–40108)
- x = a+5 (40112)
- Prepare for proc. call. Save f2's a and $ra on stack (40116–40124)
- set param ($a0) to x val (40128)
- Recover f2's a and $ra on stack (40136–40144)
- y = (result of f3(x)) (40148)
- compute val to return (y+x+a) (40152–40156)
- restore calling routine's $s0 and $s1 from stack & return (40160–40172)

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
10→5	13	5	28

$ra
40012

$pc
40160

$sp
80136

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8        # Push calling routine $s0
40104      sw $s0, 4($sp)           # and $s1 onto the stack
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5         # x = a+5
40116      addi $sp, $sp, -8        # Prepare for proc. call. Save
40120      sw $a0, 4($sp)           # f2's a and $ra on stack
40124      sw $ra, 0($sp)
40128      mov $a0, $s0             # set param ($a0) to x val
40132      jal f3
40136      lw $ra, 0($sp)           # Recover f2's a and $ra on
40140      lw $a0, 4($sp)           # stack
40144      addi $sp, $sp, 8
40148      mov $s1, $v0             # y = (result of f3(x))
40152      add $v0, $s1, Ss0        # compute val to
40156      add $v0, $v0, $a0        # return (y+x+a)
40160      lw $s0, 4($sp)           # restore calling routine's
40164      lw $s1, 0($sp)           # $s0 and $s1 from stack &
40168      addi $sp, $sp, 8         # return
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
5	13→-138	5	28

$ra
40012

$pc
40164

$sp
80136

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

- 40100–40108: Push calling routine $s0 and $s1 onto the stack
- 40112: x = a+5
- 40116–40124: Prepare for proc. call. Save f2's a and $ra on stack
- 40128: set param ($a0) to x val
- 40136–40144: Recover f2's a and $ra on stack
- 40148: y = (result of f3(x))
- 40152–40156: compute val to return (y+x+a)
- 40160–40172: restore calling routine's $s0 and $s1 from stack & return

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0	$ra	$pc	$sp
5	-138	5	28	40012	40168	80136 →80144

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

	main:	
40000	addi $s0, $zero, 5	
40004	mov $a0, $s0	
40008	jal f2	
40012	mov $s1, $v0	
40xxx	<code to print and exit>	
40100	f2: addi $sp, $sp, -8	Push calling routine $s0 and $s1 onto the stack
40104	sw $s0, 4($sp)	
40108	sw $s1, 0($sp)	
40112	addi $s0, $a0, 5	x = a+5
40116	addi $sp, $sp, -8	Prepare for proc. call. Save f2's a and $ra on stack
40120	sw $a0, 4($sp)	
40124	sw $ra, 0($sp)	
40128	mov $a0, $s0	set param ($a0) to x val
40132	jal f3	
40136	lw $ra, 0($sp)	Recover f2's a and $ra on stack
40140	lw $a0, 4($sp)	
40144	addi $sp, $sp, 8	
40148	mov $s1, $v0	y = (result of f3(x))
40152	add $v0, $s1, Ss0	compute val to return (y+x+a)
40156	add $v0, $v0, $a0	
40160	lw $s0, 4($sp)	restore calling routine's $s0 and $s1 from stack & return
40164	lw $s1, 0($sp)	
40168	addi $sp, $sp, 8	
40172	jr $ra	
40176	f3: addi $v0, $a0, 3	
40180	jr $ra	

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
5	-138	5	28

$ra
40012

$pc
40172

$sp
80144

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8       # Push calling routine $s0
40104      sw $s0, 4($sp)          # and $s1 onto the stack
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5        # x = a+5
40116      addi $sp, $sp, -8       # Prepare for proc. call.  Save
40120      sw $a0, 4($sp)          # f2's a and $ra on stack
40124      sw $ra, 0($sp)
40128      mov $a0, $s0            # set param ($a0) to x val
40132      jal f3
40136      lw $ra, 0($sp)          # Recover f2's a and $ra on
40140      lw $a0, 4($sp)          # stack
40144      addi $sp, $sp, 8
40148      mov $s1, $v0            # y = (result of f3(x))
40152      add $v0, $s1, Ss0       # compute val to
40156      add $v0, $v0, $a0       # return (y+x+a)
40160      lw $s0, 4($sp)          # restore calling routine's
40164      lw $s1, 0($sp)          # $s0 and $s1 from stack &
40168      addi $sp, $sp, 8        # return
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Non-Leaf Procedure Example

$s0	$s1	$a0	$v0
5	-138 →28	5	28

$ra
40012

$pc
40012

$sp
80144

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
      main:
40000      addi $s0, $zero, 5
40004      mov $a0, $s0
40008      jal f2
40012      mov $s1, $v0
40xxx      <code to print and exit>
40100 f2:  addi $sp, $sp, -8
40104      sw $s0, 4($sp)
40108      sw $s1, 0($sp)
40112      addi $s0, $a0, 5
40116      addi $sp, $sp, -8
40120      sw $a0, 4($sp)
40124      sw $ra, 0($sp)
40128      mov $a0, $s0
40132      jal f3
40136      lw $ra, 0($sp)
40140      lw $a0, 4($sp)
40144      addi $sp, $sp, 8
40148      mov $s1, $v0
40152      add $v0, $s1, Ss0
40156      add $v0, $v0, $a0
40160      lw $s0, 4($sp)
40164      lw $s1, 0($sp)
40168      addi $sp, $sp, 8
40172      jr $ra
40176 f3:  addi $v0, $a0, 3
40180      jr $ra
```

Push calling routine $s0 and $s1 onto the stack (40100–40108)

x = a+5 (40112)

Prepare for proc. call. Save f2's a and $ra on stack (40116–40124)

set param ($a0) to x val (40128)

Recover f2's a and $ra on stack (40136–40144)

y = (result of f3(x)) (40148)

compute val to return (y+x+a) (40152–40156)

restore calling routine's $s0 and $s1 from stack & return (40160–40172)

Non-Leaf Procedure Example

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

C code

Non-Leaf Procedure Example 2

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

C code

Push info to save onto stack

Items on stack not needed - clear them off

nested call

Need to recover original values after nested call might have changed them

```
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
```

Compiled MIPS

Non-Leaf Procedure Example 2: just stack stuff

$a0
10→3

$v0
23

$ra
40012

$pc
36000

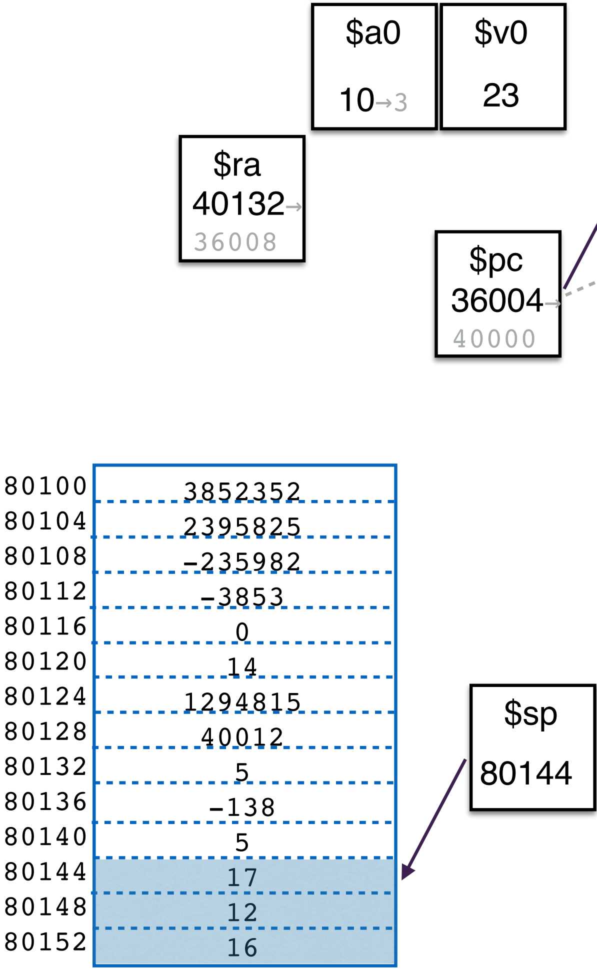
$sp
80144

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138
80140	5
80144	17
80148	12
80152	16

```
36000      addi $a0, $zero,3
36004      jal fact
36008      <print results in v0>
```

```
      fact:
40000      addi $sp, $sp, -8       # adjust stack for 2 ite
40004      sw   $ra, 4($sp)        # save return address
40008      sw   $a0, 0($sp)        # save argument
40012      slti $t0, $a0, 1        # test for n < 1
40016      beq  $t0, $zero, L1
40020      addi $v0, $zero, 1      # if so, result is 1
40024      addi $sp, $sp, 8        #   pop 2 items from sta
40028      jr   $ra                #   and return
40032 L1:  addi $a0, $a0, -1       # else decrement n
40036      jal  fact               # recursive call
40040      lw   $a0, 0($sp)        # restore original n
40044      lw   $ra, 4($sp)        #   and return address
40048      addi $sp, $sp, 8        # pop 2 items from stac
40052      mul  $v0, $a0, $v0      # multiply to get resul
40056      jr   $ra                # and return
```

Non-Leaf Procedure Example 2: just stack stuff




```
36000     addi $a0, $zero,3
36004     jal  fact
36008     <print results in v0>

        fact:
40000       addi $sp, $sp, -8        # adjust stack for 2 ite
40004       sw   $ra, 4($sp)         # save return address
40008       sw   $a0, 0($sp)         # save argument
40012       slti $t0, $a0, 1         # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1       # if so, result is 1
40024       addi $sp, $sp, 8         #   pop 2 items from sta
40028       jr   $ra                 #   and return
40032 L1:   addi $a0, $a0, -1        # else decrement n
40036       jal  fact                # recursive call
40040       lw   $a0, 0($sp)         # restore original n
40044       lw   $ra, 4($sp)         #   and return address
40048       addi $sp, $sp, 8         # pop 2 items from stac
40052       mul  $v0, $a0, $v0       # multiply to get resul
40056       jr   $ra                 # and return
```

Non-Leaf Procedure Example 2: just stack stuff

$a0	$v0
3	23

$ra
36008

$pc
40016

```
36000      addi $a0, $zero,3
36004      jal fact
36008      <print results in v0>
```

```
         fact:
40000          addi $sp, $sp, -8       # adjust stack for 2 ite
40004          sw   $ra, 4($sp)        # save return address
40008          sw   $a0, 0($sp)        # save argument
40012          slti $t0, $a0, 1        # test for n < 1
40016          beq  $t0, $zero, L1
40020          addi $v0, $zero, 1      # if so, result is 1
40024          addi $sp, $sp, 8        #   pop 2 items from sta
40028          jr   $ra                #   and return
40032    L1:   addi $a0, $a0, -1       # else decrement n
40036          jal  fact               # recursive call
40040          lw   $a0, 0($sp)        # restore original n
40044          lw   $ra, 4($sp)        #   and return address
40048          addi $sp, $sp, 8        # pop 2 items from stacl
40052          mul  $v0, $a0, $v0      # multiply to get resul
40056          jr   $ra                # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	-138 →3
80140	5 →36008
80144	17
80148	12
80152	16

$sp
80144
→80136

Non-Leaf Procedure Example 2: just stack stuff

$a0	$v0	$ra	$pc	$sp
3→2	23	36008→ 40040	40036	80136

```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>

fact:
40000     addi $sp, $sp, -8       # adjust stack for 2 ite
40004     sw   $ra, 4($sp)        # save return address
40008     sw   $a0, 0($sp)        # save argument
40012     slti $t0, $a0, 1        # test for n < 1
40016     beq  $t0, $zero, L1
40020     addi $v0, $zero, 1      # if so, result is 1
40024     addi $sp, $sp, 8        #   pop 2 items from sta
40028     jr   $ra                #   and return
40032 L1: addi $a0, $a0, -1       # else decrement n
40036     jal  fact               # recursive call
40040     lw   $a0, 0($sp)        # restore original n
40044     lw   $ra, 4($sp)        #   and return address
40048     addi $sp, $sp, 8        # pop 2 items from stac
40052     mul  $v0, $a0, $v0      # multiply to get resul
40056     jr   $ra                # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012
80132	5
80136	3
80140	36008
80144	17
80148	12
80152	16

Non-Leaf Procedure Example 2: just stack stuff

$a0	$v0
2	23

$ra
40040

$pc
40016

```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>
```

```
       fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032  L1:  addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	40012 →2
80132	5 →40040
80136	3
80140	36008
80144	17
80148	12
80152	16

$sp
80136
→80128

Non-Leaf Procedure Example 2: just stack stuff

$a0	$v0	$ra	$pc	$sp
2→1	23	40040→ 40040	40036	80218

```
36000      addi $a0, $zero,3
36004      jal fact
36008      <print results in v0>
```

```
       fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032  L1:  addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14
80124	1294815
80128	2
80132	40040
80136	3
80140	36008
80144	17
80148	12
80152	16

Non-Leaf Procedure Example 2: just stack stuff

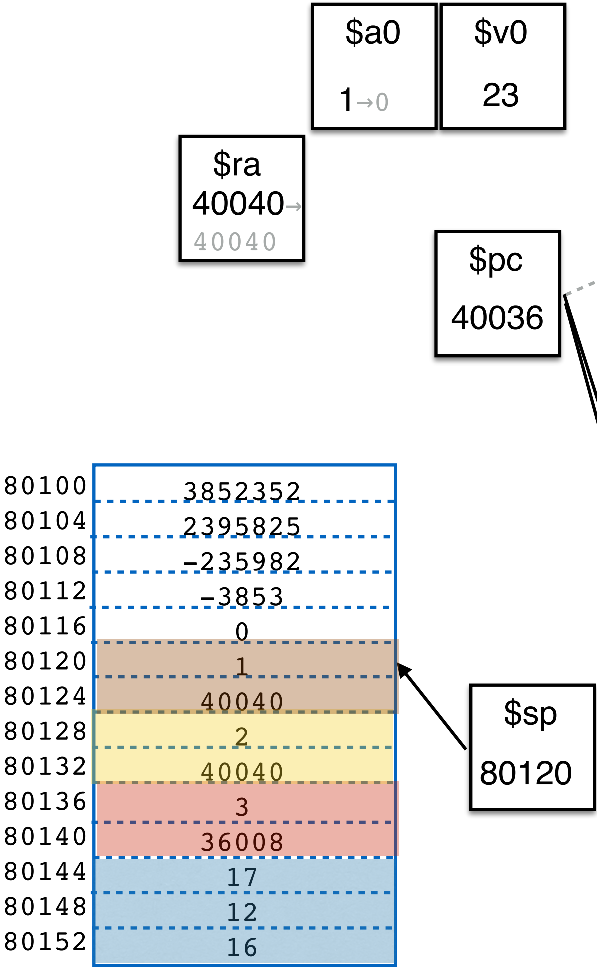
- $a0: 1
- $v0: 23
- $ra: 40040
- $pc: 40016
- $sp: 80128 →80120

```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>
```

```
      fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032 L1:   addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	-3853
80116	0
80120	14 →1
80124	1294815 →40040
80128	2
80132	40040
80136	3
80140	36008
80144	17
80148	12
80152	16

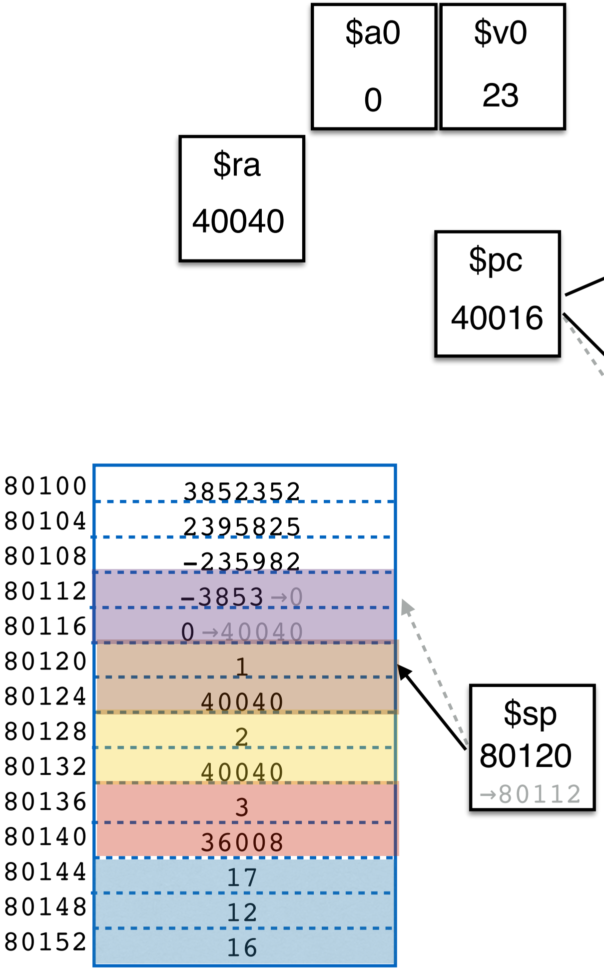
Non-Leaf Procedure Example 2: just stack stuff




```
36000      addi $a0, $zero,3
36004      jal fact
36008      <print results in v0>

          fact:
40000          addi $sp, $sp, -8      # adjust stack for 2 ite
40004          sw   $ra, 4($sp)       # save return address
40008          sw   $a0, 0($sp)       # save argument
40012          slti $t0, $a0, 1       # test for n < 1
40016          beq  $t0, $zero, L1
40020          addi $v0, $zero, 1     # if so, result is 1
40024          addi $sp, $sp, 8       #   pop 2 items from sta
40028          jr   $ra               #   and return
40032     L1:  addi $a0, $a0, -1      # else decrement n
40036          jal  fact              # recursive call
40040          lw   $a0, 0($sp)       # restore original n
40044          lw   $ra, 4($sp)       #   and return address
40048          addi $sp, $sp, 8       # pop 2 items from stac
40052          mul  $v0, $a0, $v0     # multiply to get resul
40056          jr   $ra               # and return
```

Non-Leaf Procedure Example 2: just stack stuff




```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>

        fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032   L1: addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

Non-Leaf Procedure Example 2: just stack stuff

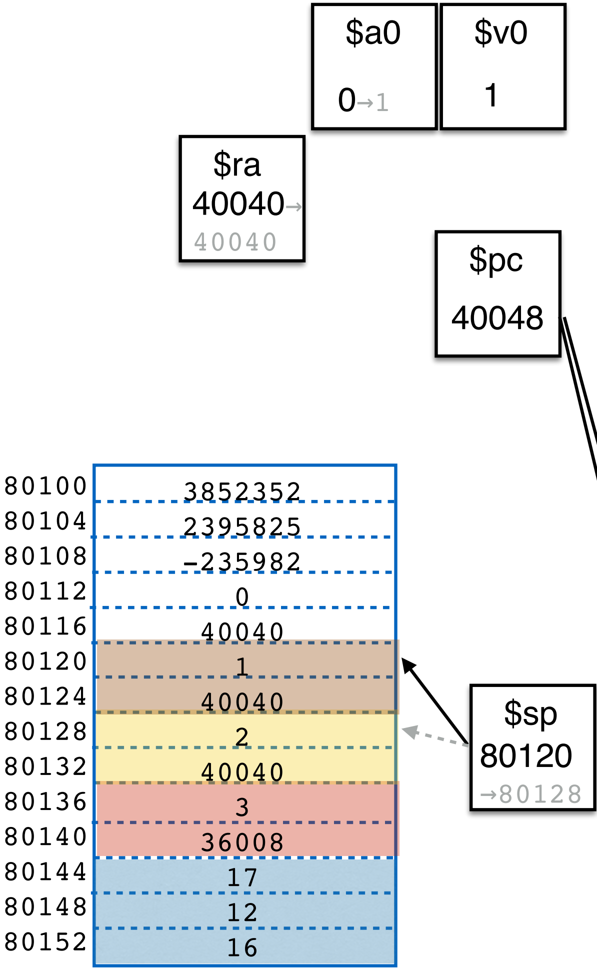
$a0	$v0	$ra	$pc	$sp
0	23→1	40040	40028	80112 →80120

```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>
```

```
      fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032 L1:   addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	0
80116	40040
80120	1
80124	40040
80128	2
80132	40040
80136	3
80140	36008
80144	17
80148	12
80152	16

Non-Leaf Procedure Example 2: just stack stuff




```
36000      addi $a0, $zero,3
36004      jal fact
36008      <print results in v0>

       fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032  L1:  addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

Non-Leaf Procedure Example 2: just stack stuff

$a0	$v0	$ra	$pc	$sp
1	1→1	40040	40056	80128

```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>

        fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032   L1: addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	0
80116	40040
80120	1
80124	40040
80128	2
80132	40040
80136	3
80140	36008
80144	17
80148	12
80152	16

Non-Leaf Procedure Example 2: just stack stuff

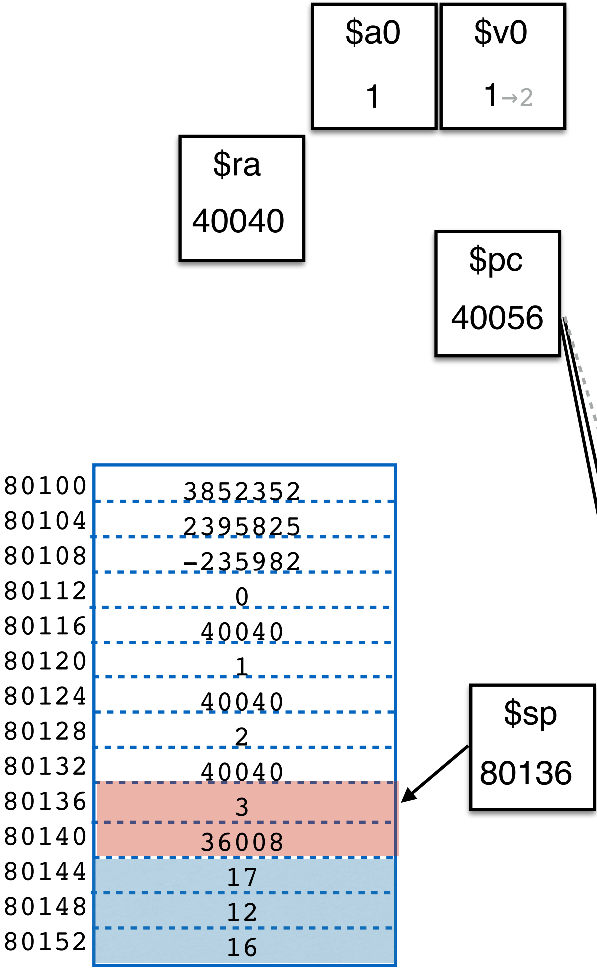
$a0	$v0	$ra	$pc	$sp
1 →2	1	40040→ 40040	40048	80128 →80136

```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>

      fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 ite
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from sta
40028       jr   $ra                #   and return
40032 L1:   addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stac
40052       mul  $v0, $a0, $v0      # multiply to get resul
40056       jr   $ra                # and return
```

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	0
80116	40040
80120	1
80124	40040
80128	2
80132	40040
80136	3
80140	36008
80144	17
80148	12
80152	16

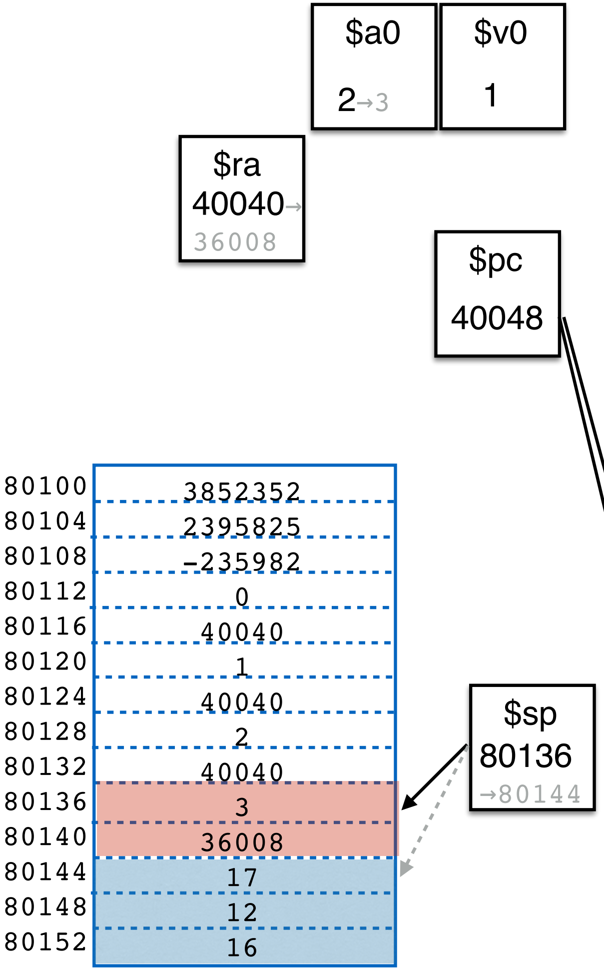
Non-Leaf Procedure Example 2: just stack stuff




```
36000      addi $a0, $zero,3
36004      jal fact
36008      <print results in v0>

           fact:
40000          addi $sp, $sp, -8       # adjust stack for 2 ite
40004          sw   $ra, 4($sp)        # save return address
40008          sw   $a0, 0($sp)        # save argument
40012          slti $t0, $a0, 1        # test for n < 1
40016          beq  $t0, $zero, L1
40020          addi $v0, $zero, 1      # if so, result is 1
40024          addi $sp, $sp, 8        #   pop 2 items from sta
40028          jr   $ra                #   and return
40032      L1: addi $a0, $a0, -1       # else decrement n
40036          jal  fact               # recursive call
40040          lw   $a0, 0($sp)        # restore original n
40044          lw   $ra, 4($sp)        #   and return address
40048          addi $sp, $sp, 8        # pop 2 items from stac
40052          mul  $v0, $a0, $v0      # multiply to get resul
40056          jr   $ra                # and return
```

Non-Leaf Procedure Example 2: just stack stuff




```
36000     addi $a0, $zero,3
36004     jal fact
36008     <print results in v0>

      fact:
40000      addi $sp, $sp, -8      # adjust stack for 2 ite
40004      sw   $ra, 4($sp)       # save return address
40008      sw   $a0, 0($sp)       # save argument
40012      slti $t0, $a0, 1       # test for n < 1
40016      beq  $t0, $zero, L1
40020      addi $v0, $zero, 1     # if so, result is 1
40024      addi $sp, $sp, 8       #   pop 2 items from sta
40028      jr   $ra               #   and return
40032 L1:  addi $a0, $a0, -1      # else decrement n
40036      jal  fact              # recursive call
40040      lw   $a0, 0($sp)       # restore original n
40044      lw   $ra, 4($sp)       #   and return address
40048      addi $sp, $sp, 8       # pop 2 items from stac
40052      mul  $v0, $a0, $v0     # multiply to get result
40056      jr   $ra               # and return
```

Non-Leaf Procedure Example 2: just stack stuff

$a0	$v0	$ra	$pc	$sp
3	2→6	36008	40056	80144

```
36000    addi $a0, $zero,3
36004    jal fact
36008    <print results in v0>

fact:
40000      addi $sp, $sp, -8      # adjust stack for 2 ite
40004      sw   $ra, 4($sp)       # save return address
40008      sw   $a0, 0($sp)       # save argument
40012      slti $t0, $a0, 1       # test for n < 1
40016      beq  $t0, $zero, L1
40020      addi $v0, $zero, 1     # if so, result is 1
40024      addi $sp, $sp, 8       #   pop 2 items from sta
40028      jr   $ra               #   and return
40032 L1:  addi $a0, $a0, -1      # else decrement n
40036      jal  fact              # recursive call
40040      lw   $a0, 0($sp)       # restore original n
40044      lw   $ra, 4($sp)       #   and return address
40048      addi $sp, $sp, 8       # pop 2 items from stac
40052      mul  $v0, $a0, $v0     # multiply to get resul
40056      jr   $ra               # and return
```

Address	Value
80100	3852352
80104	2395825
80108	-235982
80112	0
80116	40040
80120	1
80124	40040
80128	2
80132	40040
80136	3
80140	36008
80144	17
80148	12
80152	16

Non-Leaf Procedure Example 2: just stack stuff

$a0	$v0	$ra	$pc	$sp
3	6	36008	36008	80144

```
36000    addi $a0, $zero,3
36004    jal  fact
36008    <print results in v0>
```

```
      fact:
40000      addi $sp, $sp, -8      # adjust stack for 2 ite
40004      sw   $ra, 4($sp)       # save return address
40008      sw   $a0, 0($sp)       # save argument
40012      slti $t0, $a0, 1       # test for n < 1
40016      beq  $t0, $zero, L1
40020      addi $v0, $zero, 1     # if so, result is 1
40024      addi $sp, $sp, 8       #   pop 2 items from sta
40028      jr   $ra               #   and return
40032 L1:  addi $a0, $a0, -1      # else decrement n
40036      jal  fact              # recursive call
40040      lw   $a0, 0($sp)       # restore original n
40044      lw   $ra, 4($sp)       #   and return address
40048      addi $sp, $sp, 8       # pop 2 items from stac
40052      mul  $v0, $a0, $v0     # multiply to get resul
40056      jr   $ra               # and return
```

80100	3852352
80104	2395825
80108	-235982
80112	0
80116	40040
80120	1
80124	40040
80128	2
80132	40040
80136	3
80140	36008
80144	17
80148	12
80152	16

MIPS Errata

Sign-extend

- Suppose we want to convert an unsigned 8-bit number to its equivalent unsigned 16-bit number.
    - e.g., extend 10110110
    - Prepend 8 0's: 00000000 10110110
- Suppose we want to convert a signed (2's complement) 8-bit number to its equivalent unsigned 16-bit number
    - e.g., extend 10110110
    - It's negative: turns out prepend all 1's to extend: 11111111 10110110
    - If it's positive, prepend 0's, e.g., 00001010 extends to 00000000 00001010
- So for unsigned, always prepend 0's to extend. For 2's complement, prepend the high-order bit: this is called sign-extend

Partial-Word memory Ops

Memory Byte/Halfword Operations

- Could use bitwise operations
- MIPS has byte/halfword load/store FROM/TO MEMORY (note offset and rs value need not be a multiple of 4)
  - `lb rt, offset(rs)  # sign extend byte to 32 bits in rt`
  - `lh rt, offset(rs)  # sign extend halfword to 32 bits in rt`
  - `lbu rt, offset(rs) # zero extend byte to 32 bits in rt`
  - `lhu rt, offset(rs) # zero extend halfword to 32 bits in rt`
  - `sb rt, offset(rs)  # store rightmost byte of rt`
  - `sh rt, offset(rs)  # store rightmost halfword of rt`

String Copy Example

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

C code (naive)

- Null-terminated string, each item in array is a byte
- Addresses of x and y in \$a0 and \$a1 respectively
- i in \$s0

String Copy Example

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

C code (naive)

Assume:
- addr of x passed in through $a0
- addr of y passed in through $a1
- i to be stored in $s0

```
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
```

Compiled MIPS

Constants and their (bit) size

16-bit constants

- MIPS requires use of 16-bit constants (instead of 32-bit)
- Why?
  - Constants often embedded in 32-bit instruction
  - Some of those bits have to describe the instruction: only a subset of bits can be devoted to the constant
  - So: use 16 bits to describe instruction, 16 bits remain to describe (value of) constant

32-bit constants

- Most constants are small, 16 bits usually sufficient

- Occasionally, need 32-bit constant in memory location

```
lui rt, constant
```

  - copies 16-bit constant to the left (upper) bits of rt

  - clears right (lower) 16 bits of rt to 0

- example usage: copy 4,000,000 (= 61 * 2^{16} + 2304) into $s0: takes 2 instructions

```
    lui $s0, 61       $s0: 0000 0000 0111 1101 0000 0000 0000 0000
ori $s0, $s0, 2304    $s0: 0000 0000 0111 1101 0000 1001 0000 0000
```

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions, one to one.

- Pseudoinstructions are shorthand. They are recognized by the assembler but translated into small bundles of machine instructions.

```
move $t0,$t1      --becomes-->  add $t0,$zero,$t1

blt $t0,$t1,L     --becomes-->  slt $at,$t0,$t1
                                bne $at,$zero,L
```

- `$at` (register 1) is an "assembler temporary"

Programming Pitfalls

- Sequential words are not at sequential addresses -- increment by 4 not by 1!

- Keeping a pointer to an automatic variable (on the stack) after procedure returns