CS 110
Computer Architecture
Lecture 6:
More MIPS, MIPS Functions

Instructor:
Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

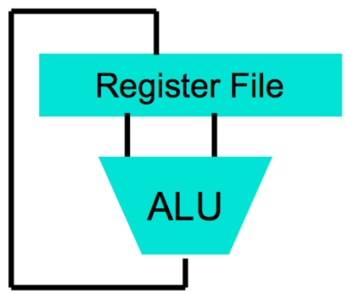
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

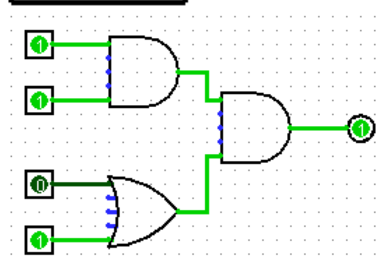
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

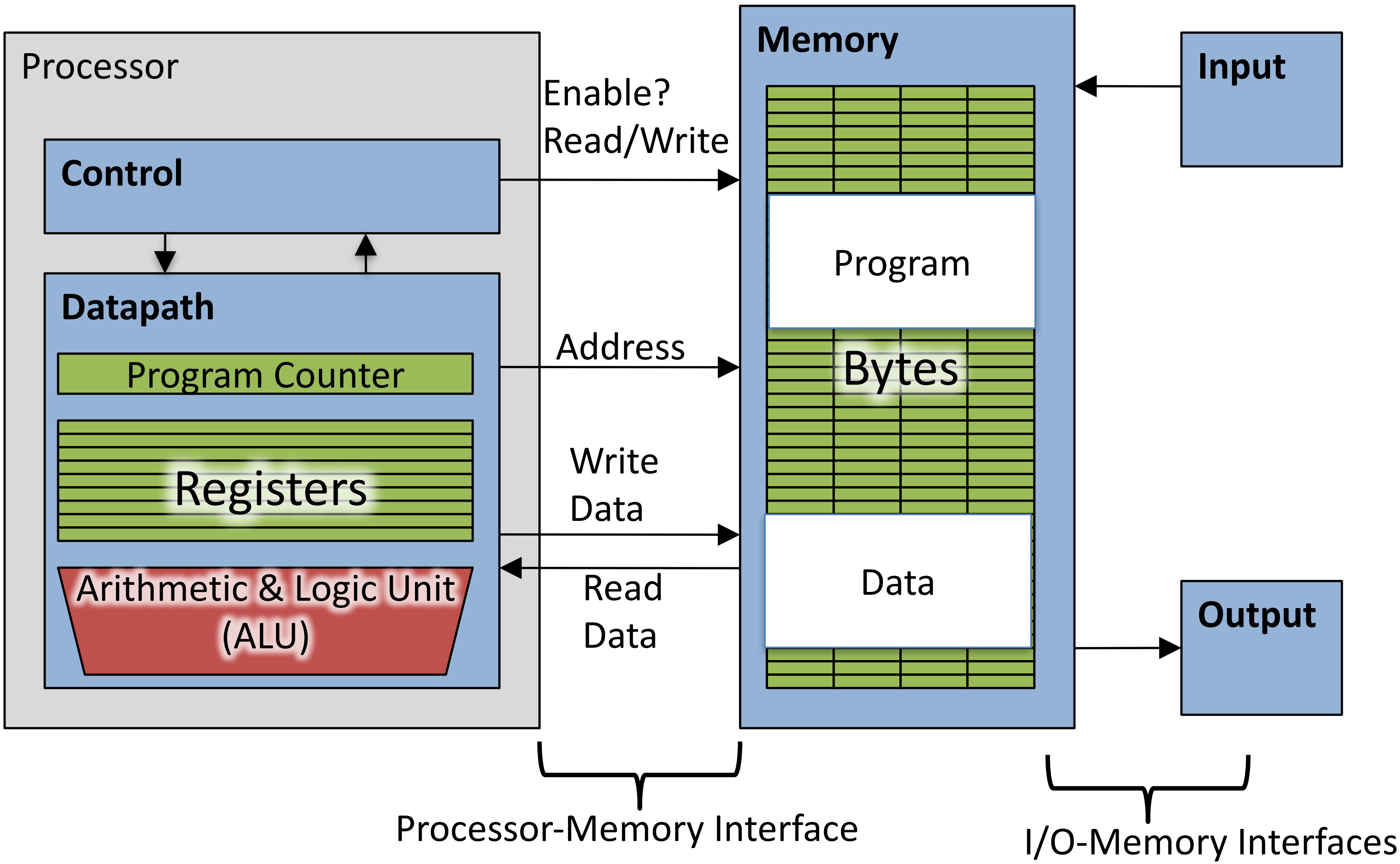
Logic Circuit Description (Circuit Schematic Diagrams)



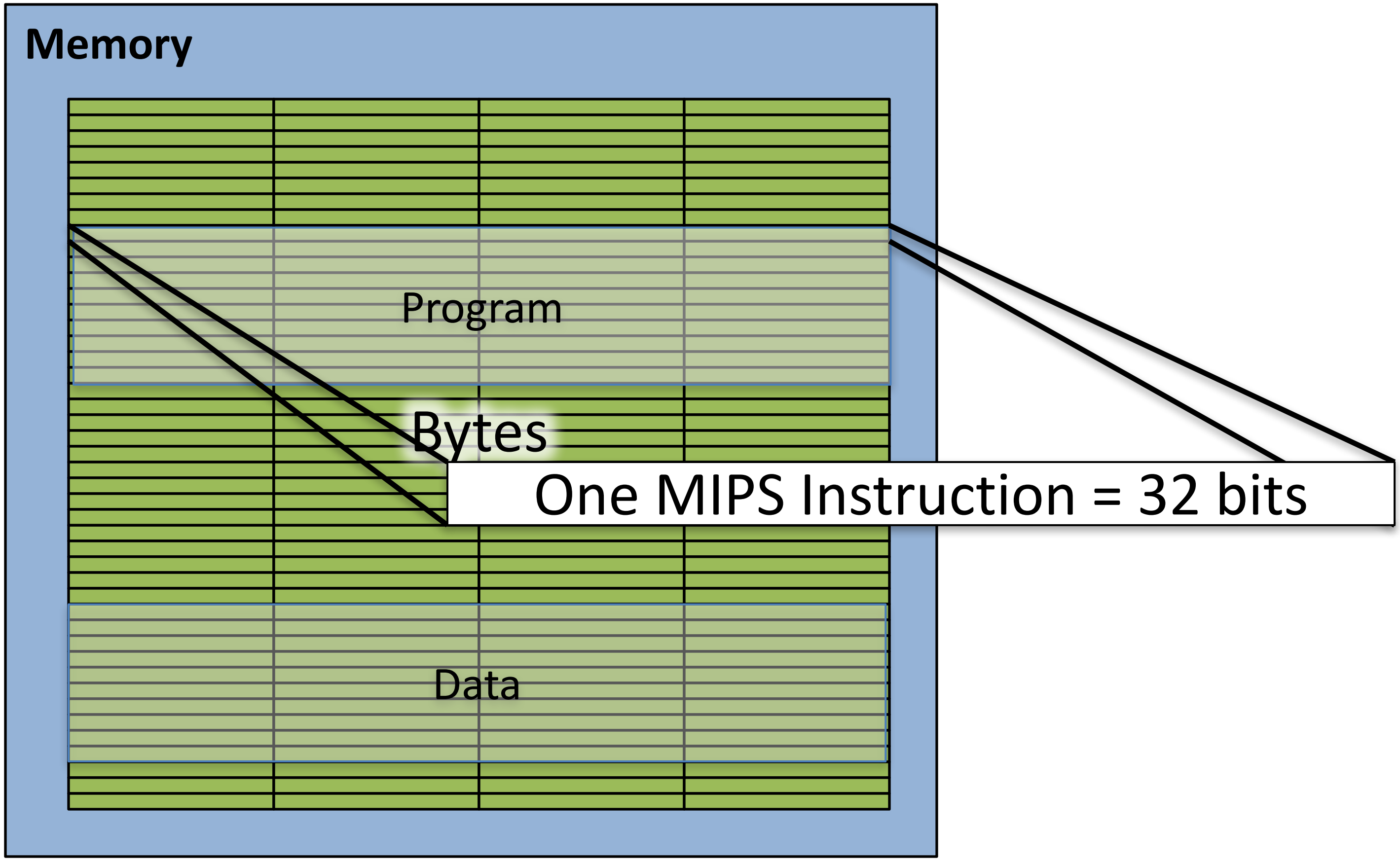
From last lecture …

- Computer "words" and "vocabulary" are called *instructions* and *instruction set* respectively
- MIPS is example RISC instruction set used here
- Rigid format: 1 operation, 2 source operands, 1 destination
  - `add,sub,mul,div,and,or,sll,srl,sra`
  - `lw,sw,lb,sb` to move data to/from registers from/to memory
  - `beq, bne, j, slt, slti` for decision/flow control
- Simple mappings from arithmetic expressions, array access, in C to MIPS instructions

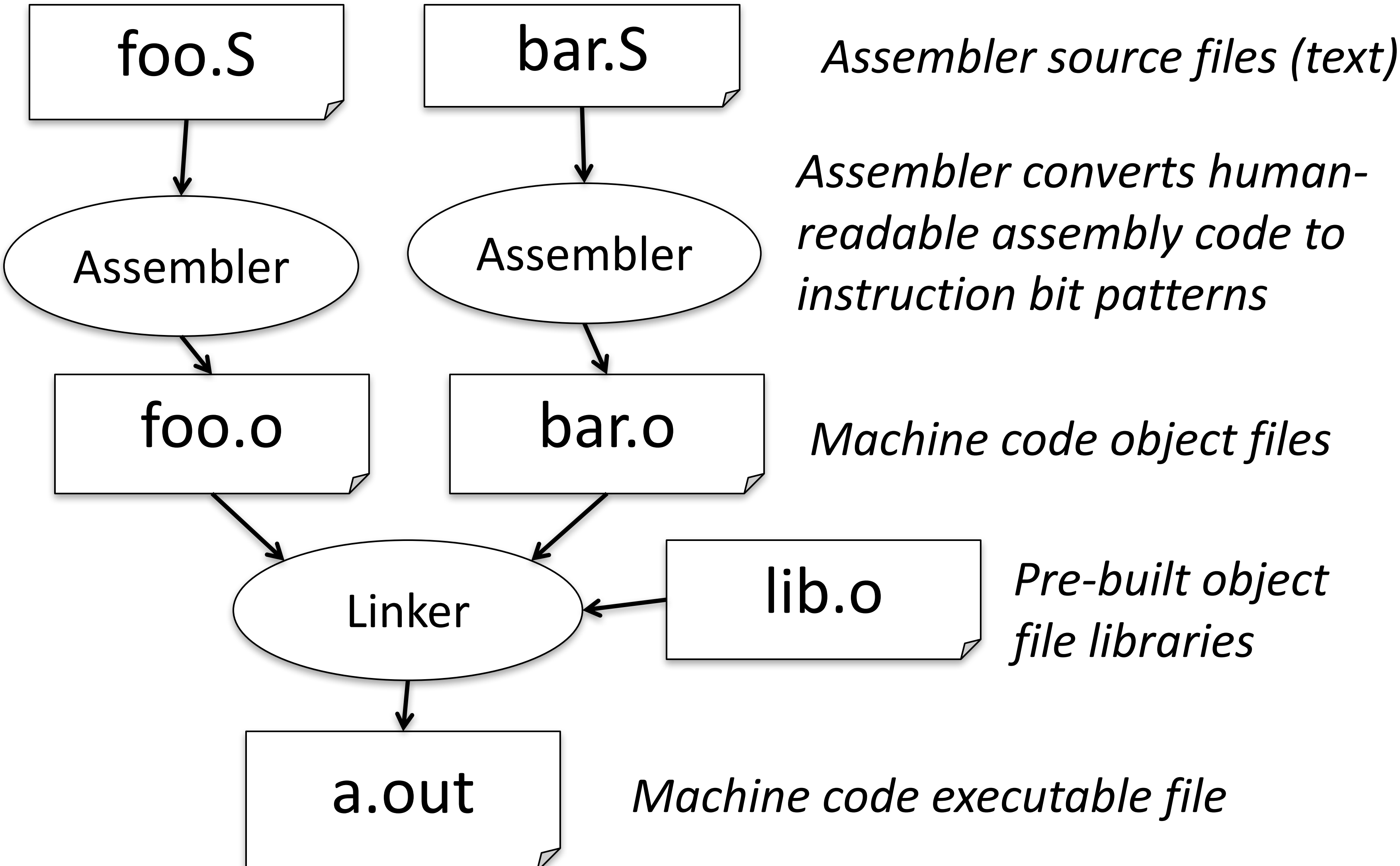
Review: Components of a Computer



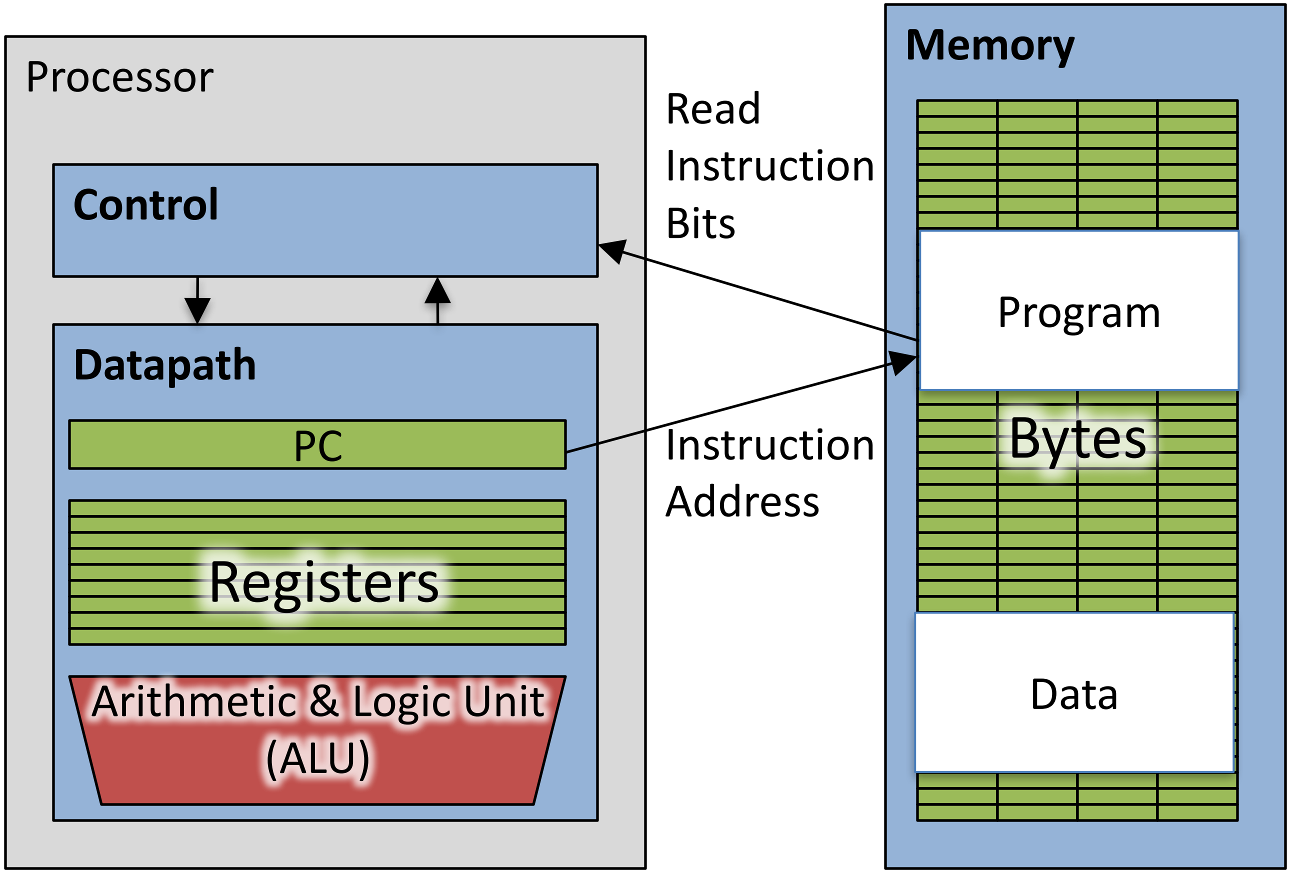
How Program is Stored



Assembler to Machine Code (more later in course)



Executing a Program



- The PC (program counter) is internal register inside processor holding <u>byte</u> address of next instruction to be executed.
- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates program counter (default is <u>add +4 bytes to PC</u>, to move to next sequential instruction)

Review *if-else* Statement

- Assuming translations below, compile

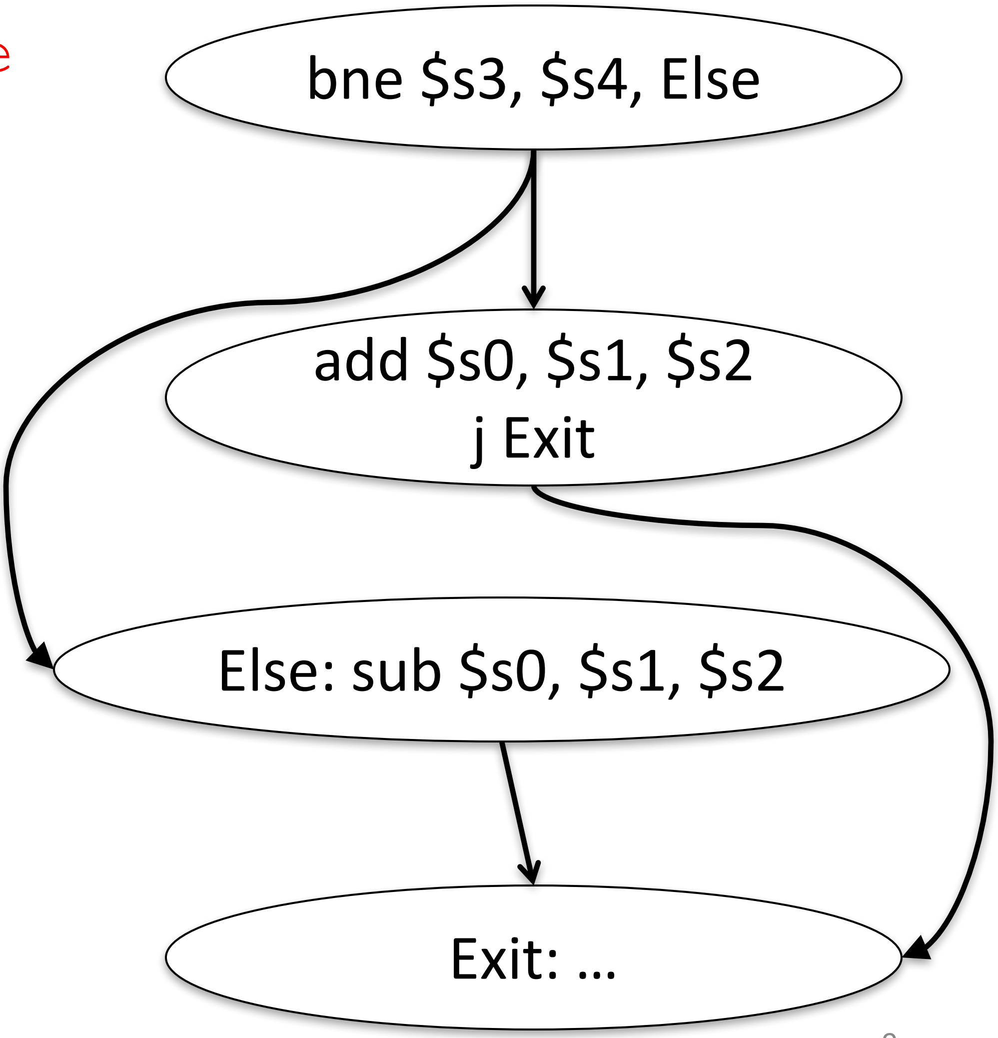
  f → `$s0` g → `$s1` h → `$s2`

  i → `$s3` j → `$s4`

```
if (i == j)                bne $s3,$s4,Else
  f = g + h;               add $s0,$s1,$s2
else                       j Exit
  f = g - h;         Else: sub $s0,$s1,$s2
                     Exit:
```

Control-flow Graphs: A visualization

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



Question!

```
        addi $s0,$zero,0
Start:  slt  $t0,$s0,$s1
        beq  $t0,$zero,Exit
        sll  $t1,$s0,2
        addu $t1,$t1,$s5
        lw   $t1,0($t1)
        add  $s4,$s4,$t1
        addi $s0,$s0,1
        j Start
Exit:
```

What is the code above?

A: while loop

B: do … while loop

C: for loop

D: A or C

E: Not a loop

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling code can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

MIPS Function Call Conventions

- Registers faster than memory, so use them
- `$a0–$a3`: four *argument* registers to pass parameters ($4 - $7)
- `$v0,$v1`: two *value* registers to return values ($2,$3)
- `$ra`: one *return address* register to return to the point of origin ($31)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
 }
 int sum(int x, int y) {
   return x+y;
 }
```

MIPS

```
address (shown in decimal)
 1000
 1004
 1008
 1012
 1016
 …
 2000
 2004
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
 }
 int sum(int x, int y) {
   return x+y;
 }
```

MIPS

```
address (shown in decimal)
 1000 add  $a0,$s0,$zero   # x = a
 1004 add  $a1,$s1,$zero   # y = b
 1008 addi $ra,$zero,1016  # $ra=1016
 1012 j    sum             # jump to sum
 1016 …                    # next instruction
 …
 2000 sum: add $v0,$a0,$a1
 2004 jr   $ra    # new instr. "jump register"
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
  return x+y;
}
```

MIPS

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

```
2000 sum: add $v0,$a0,$a1
2004 jr   $ra # new instr. "jump register"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (**`jal`**)
- Before:

```
1008 addi $ra,$zero,1016  # $ra=1016
1012 j sum                # goto sum
```

- After:

```
1008 jal sum  # $ra=1012,goto sum
```

- Why have a **`jal`**?
  - Make the common case fast: function calls very common.
  - Don’t have to know where code is in memory with **`jal`**!

MIPS Function Call Instructions

- Invoke function: *jump and link* instruction (`jal`) (really should be `laj` *"link and jump"*)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the following instruction in register `$ra ($31)`

```
jal FunctionLabel
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register

```
jr $ra
```

Notes on Functions

- Calling program (*caller*) puts parameters into registers `$a0-$a3` and uses `jal X` to invoke (*callee*) at address labeled X
- Must have register in computer with address of currently executing instruction
  - Instead of *Instruction Address Register* (better name), historically called *Program Counter* (*PC*)
  - It's a program's counter; it doesn't count programs!

- What value does `jal X` place into `$ra`? **????**
- `jr $ra` puts address inside `$ra` back into PC

Where Are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before call function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- `$sp` is the *stack pointer* in MIPS ($29)
- Convention is grow from high to low addresses
  - *Push* decrements `$sp`, *Pop* increments `$sp`

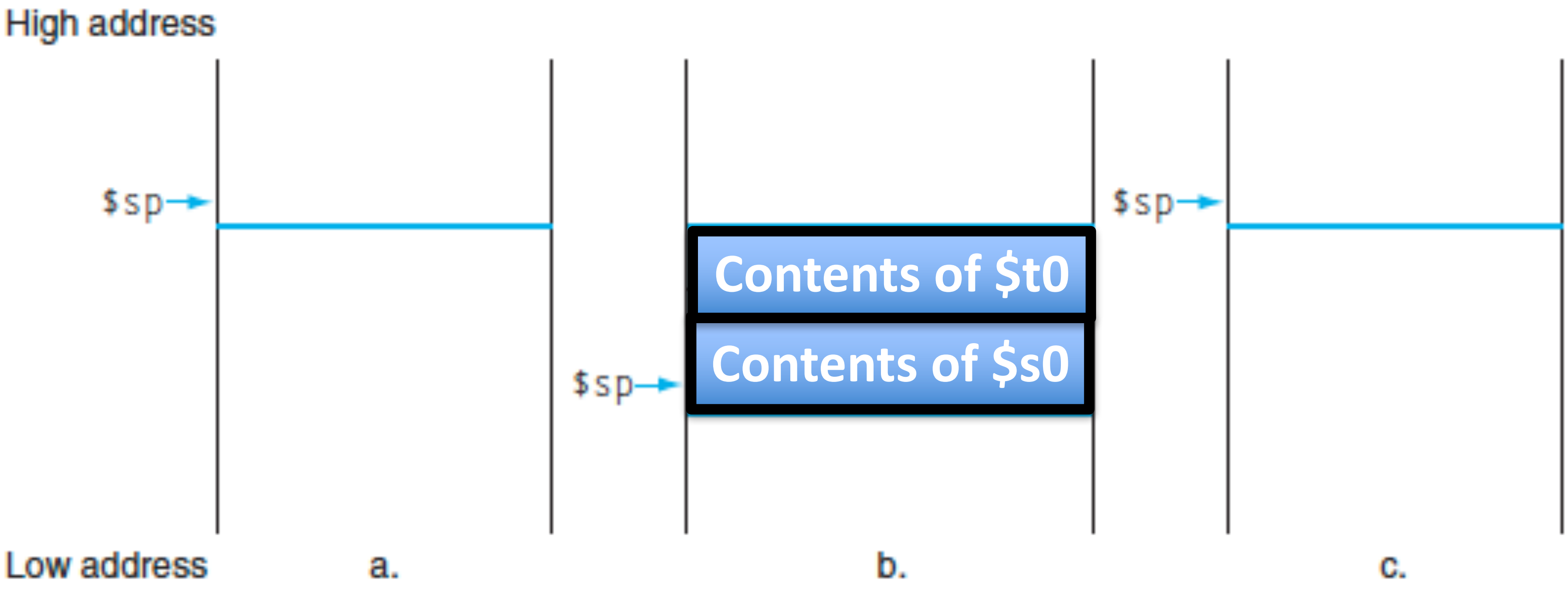
Example

```
int Leaf
  (int g, int h, int i, int j)
{
  int f;
  f = (g + h) – (i + j);
  return f;
}
```

- Parameter variables `g`, `h`, `i`, and `j` in argument registers `$a0`, `$a1`, `$a2`, and `$a3`, and `f` in `$s0`
- Assume need one temporary register `$t0`

Stack Before, During, After Function

- Need to save old values of `$s0` and `$t0`



MIPS Code for Leaf()

```
Leaf: addi  $sp,$sp,-8     # adjust stack for 2 items
      sw    $t0, 4($sp)    # save $t0 for use afterwards
      sw    $s0, 0($sp)    # save $s0 for use afterwards

      add   $s0,$a0,$a1    # f = g + h
      add   $t0,$a2,$a3    # t0 = i + j
      sub   $v0,$s0,$t0    # return value (g + h) – (i + j)

      lw    $s0, 0($sp)    # restore register $s0 for caller
      lw    $t0, 4($sp)    # restore register $t0 for caller
      addi  $sp,$sp,8      # adjust stack to delete 2 items
      jr    $ra            # jump back to calling routine
```

Administrivia

- HW1 was due yesterday – don't push after the deadline to avoid automatic use of slip days (unless you have to)
- HW2 has been posted – due next Wednesday
- Use of slipdays…
  - Project team number of slip days = min(#a, #b)
- Potentially the discussion location will be changed to SIST 1A-200!

What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in `$a0` to `$a3` and `$ra`
- What is the solution?

Nested Procedures (1/2)

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;
}
```

- Something called **sumSquare**, now **sumSquare** is calling **mult**
- So there's a value in $ra that **sumSquare** wants to jump back to, but this will be overwritten by the call to **mult**

Need to save **sumSquare** return address before call to **mult**

Nested Procedures (2/2)

- In general, may need to save some other info in addition to `$ra`.
- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
  - Heap: Variables declared dynamically via **malloc**
  - Stack: Space to be used by procedure during execution; this is where we can save register values

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - `$sp`, `$gp`, `$fp`, "saved registers" `$s0`- `$s7`
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Return value registers `$v0`,`$v1`, Argument registers `$a0`-`$a3`, "temporary registers" `$t0`-`$t9`, `$ra`

Load Halfword Unsigned	`lhu`	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25$_{hex}$
Load Linked	`ll`	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30$_{hex}$
Load Upper Imm.	`lui`	I	R[rt] = {imm, 16'b0}		f$_{hex}$
Load Word	`lw`	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23$_{hex}$
Nor	`nor`	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27$_{hex}$
Or	`or`	R	R[rd] = R[rs] | R[rt]		0 / 25$_{hex}$
Or Immediate	`ori`	I	R[rt] = R[rs] | ZeroExtImm	(3)	d$_{hex}$
Set Less Than	`slt`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / 2a$_{hex}$
Set Less Than Imm.	`slti`	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a$_{hex}$
Set Less Than Imm. Unsigned	`sltiu`	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b$_{hex}$
Set Less Than Unsig.	`sltu`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / 2b$_{hex}$
Shift Left Logical	`sll`	R	R[rd] = R[rt] << shamt		0 / 00$_{hex}$
Shift Right Logical	`srl`	R	R[rd] = R[rt] >> shamt		0 / 02$_{hex}$
Store Byte	`sb`	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28$_{hex}$
Store Conditional	`sc`	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38$_{hex}$
Store Halfword	`sh`	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29$_{hex}$
Store Word	`sw`	I	M[R[rs]+SignExtImm] = R[rt]	(2)	2b$_{hex}$
Subtract	`sub`	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22$_{hex}$
Subtract Unsigned	`subu`	R	R[rd] = R[rs] - R[rt]		0 / 23$_{hex}$

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R

opcode	rs	rt	rd	shamt	funct
31 26	25 21	20 16	15 11	10 6	5 0

I

opcode	rs	rt	immediate
31 26	25 21	20 16	15 0

J

opcode	address
31 26	25 0

Load FP Double	`ldc1`	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	`mfhi`	R	R[rd] = Hi		0 /--/--/10
Move From Lo	`mflo`	R	R[rd] = Lo		0 /--/--/12
Move From Control	`mfc0`	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	`mult`	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	`multu`	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	`sra`	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	`swc1`	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	`sdc1`	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR

opcode	fmt	ft	fs	fd	funct
31 26	25 21	20 16	15 11	10 6	5 0

FI

opcode	fmt	ft	immediate
31 26	25 21	20 16	15 0

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	`blt`	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	`bgt`	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	`ble`	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	`bge`	if(R[rs]>=R[rt]) PC = Label
Load Immediate	`li`	R[rd] = immediate
Move	`move`	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

Question

- Which statement is FALSE?

A: MIPS uses `jal` to invoke a function and jr to return from a function

B: `jal` saves PC+1 in `$ra`

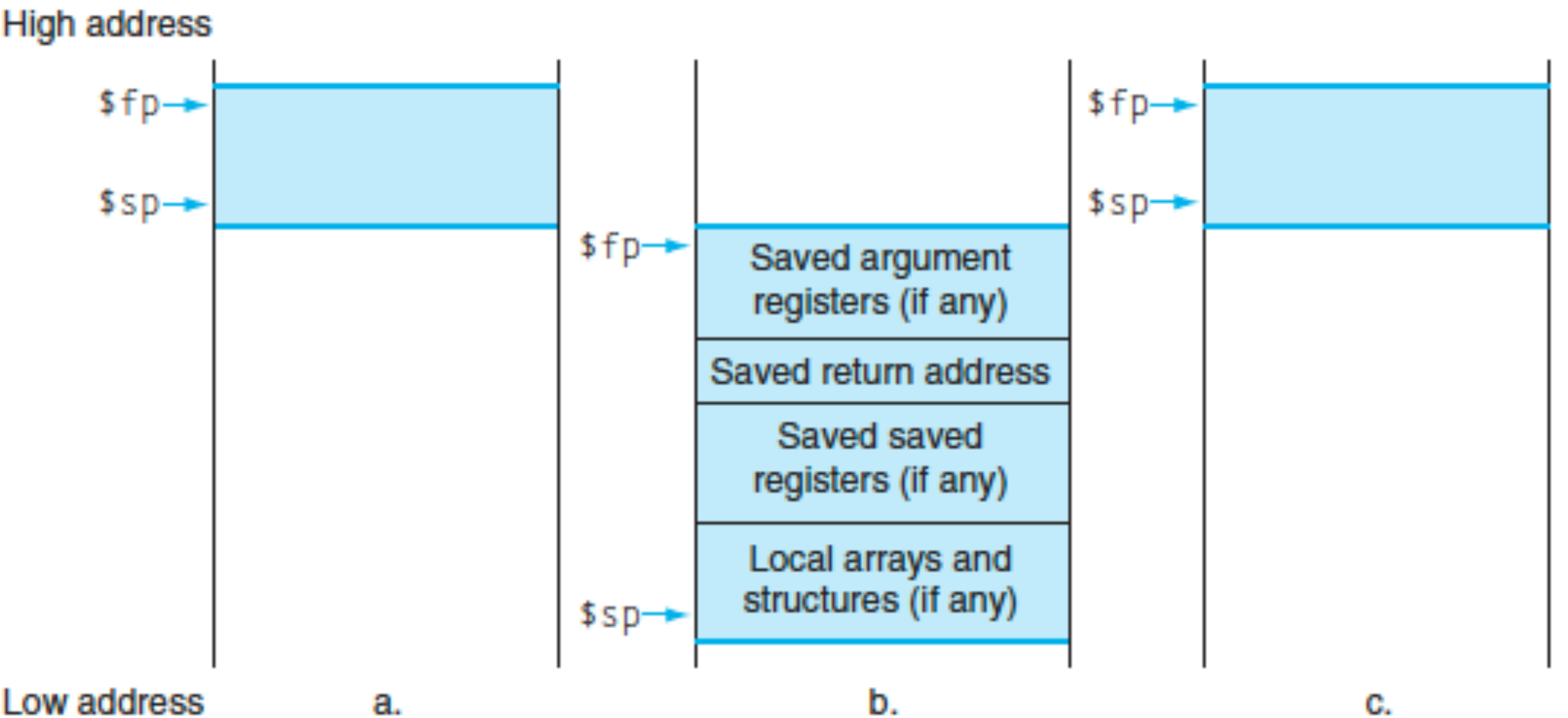
C: The callee can use temporary registers (`$ti`) without saving and restoring them

D: The caller can rely on save registers (`$si`) without fear of callee changing them

Allocating Space on Stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits
  - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don't fit in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer (`$fp`) to point to first word of frame

Stack Before, During, After Call



Using the Stack (1/2)

- So we have a register **`$sp`** which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

Using the Stack (2/2)

- Hand-compile

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

“push”

“pop”

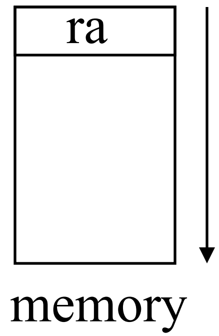
```
sumSquare:
        addi  $sp,$sp,-8      # space on stack
        sw    $ra, 4($sp)     # save ret addr
        sw    $a1, 0($sp)     # save y
        add   $a1,$a0,$zero   # mult(x,x)
        jal   mult            # call mult
        lw    $a1, 0($sp)     # restore y
        add   $v0,$v0,$a1     # mult()+y
        lw    $ra, 4($sp)     # get ret addr
        addi  $sp,$sp,8       # restore stack
        jr    $ra
mult:   ...
```

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be
```

Body ··· **(call other functions…)**



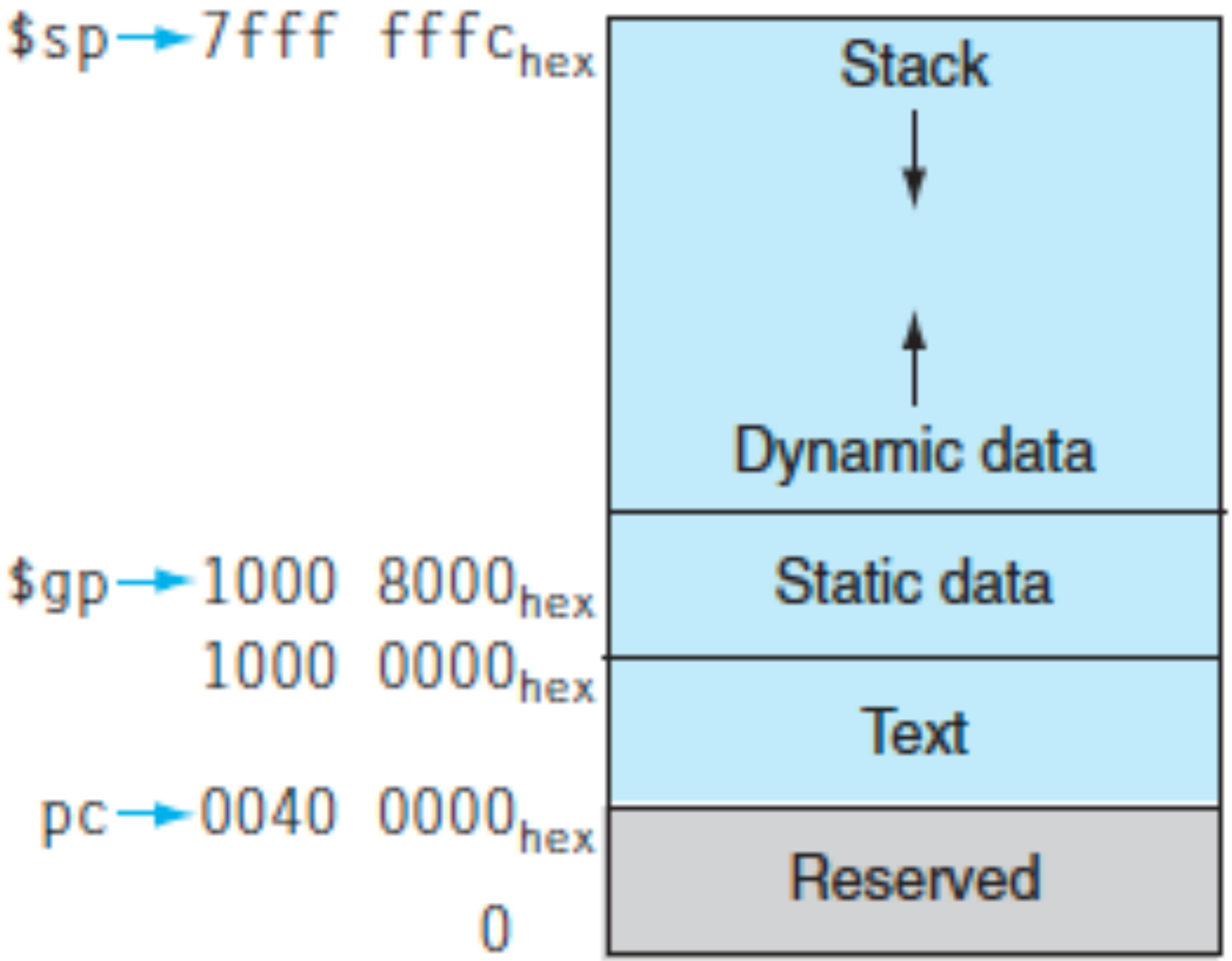
Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```

Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : 7fff fffc_{hex}
- MIPS programs (*text segment*) in low end
  - 0040 0000_{hex}
- *static data segment* (constants and other static variables) above text for static variables
  - MIPS convention *global pointer* (`$gp`) points to static
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation



Register Allocation and Numbering

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

And in Conclusion…

- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…

  Arithmetic: `add, addi, sub, addu, addiu, subu`

  Memory: `lw, sw, lb, sb`

  Decision: `beq, bne, slt, slti, sltu, sltiu`

  Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them!
  - \$a0-\$a3 for function arguments, \$v0-\$v1 for return values
  - \$sp, stack pointer, \$fp frame pointer, \$ra return address

Bonus Slides

Recursive Function Factorial

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

Recursive Function Factorial

```
Fact:
   # adjust stack for 2 items
   addi $sp,$sp,-8
   # save return address
   sw $ra, 4($sp)
   # save argument n
   sw $a0, 0($sp)
   # test for n < 1
   slti $t0,$a0,1
   # if n >= 1, go to L1
   beq $t0,$zero,L1
   # Then part (n==1) return 1
   addi $v0,$zero,1
   # pop 2 items off stack
   addi $sp,$sp,8
   # return to caller
   jr $ra
```

```
L1:
   # Else part (n >= 1)
   # arg. gets (n – 1)
   addi $a0,$a0,-1
   # call fact with (n – 1)
   jal Fact
   # return from jal: restore n
   lw $a0, 0($sp)
   # restore return address
   lw $ra, 4($sp)
   # adjust sp to pop 2 items
   addi $sp, $sp,8
   # return n * fact (n – 1)
   mul $v0,$a0,$v0
   # return to the caller
   jr $ra
```

mul is a pseudo instruction