Systems Architecture

Compilers, Assemblers, Linkers & Loaders

Jeremy R. Johnson

Anatole D. Ruslanov

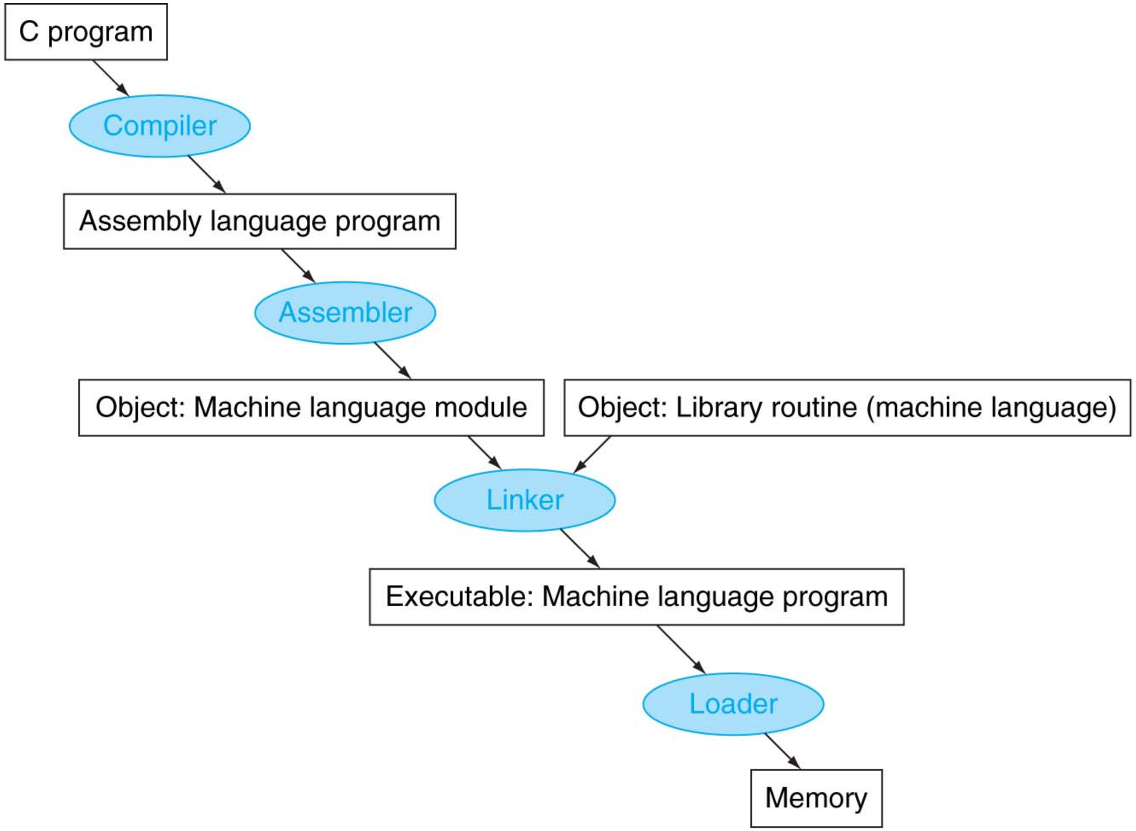
William M. Mongan

Some material drawn from CMU CSAPP Slides: Kesden and Puschel

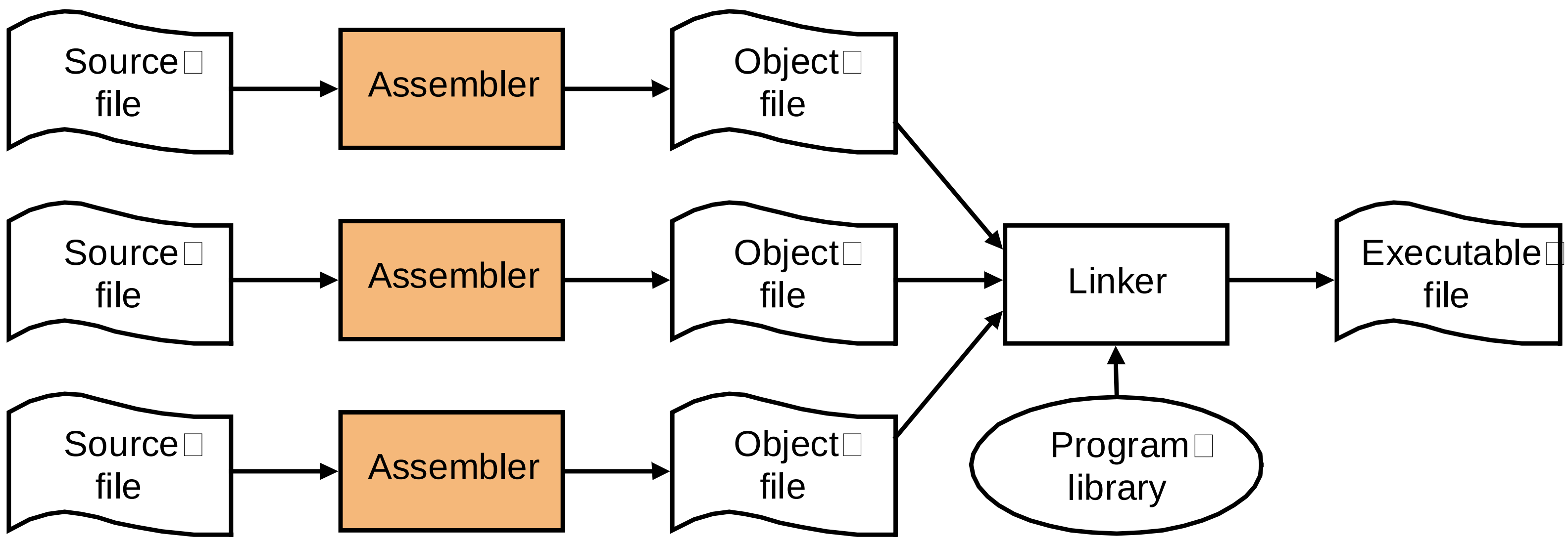
Introduction

- **Objective: To introduce the role of compilers, assemblers, linkers and loaders. To see what is underneath a C program: assembly language, machine language, and executable.**

Compilation Process



Compilation Process



Below Your Program

Example from a Unix system

- **Source Files: count.c and main.c**
- **Corresponding assembly code: count.s and main.s**
- **Corresponding machine code (object code): count.o and main.o**
- **Library functions: libc.a**
- **Executable file: a.out**

- **format for a.out and object code:**
  ELF (Executable and Linking Format)

Producing an Executable Program

Example from a Unix system (SGI Challenge running IRIX 6.5)

- **Compiler: count.c and main.c → count.s and main.s**
  - gcc -S count.c main.c
- **Assembler: count.s and main.s → count.o and main.o**
  - gcc -c count.s main.s
  - as count.s -o count.o
- **Linker/Loader: count.o main.o libc.a → a.out**
  - gcc main.o count.o
  - ld main.o count.o -lc (additional libraries are required)

Source Files

```
void main()
{
  int n,s;

  printf("Enter upper limit:  ");
  scanf("%d",&n);
  s = count(n);
  printf("Sum of i from 1 to %d =
      %d\n",n,s);
}
```

```
int count(int n)
{
  int i,s;

  s = 0;
  for (i=1;i<=n;i++)
    s = s + i;
  return s;
}
```

Assembly Code for MIPS (count.s)

```
#.file  1 "count.c"
        .option pic2
        .section        .text
        .text
        .align  2
        .globl  count
        .ent    count
count:
.LFB1:
        .frame  $fp,48,$31          # vars= 16, regs= 2/0, args= 0, extra= 1
6
        .mask   0x50000000,-8
        .fmask  0x00000000,0
        subu    $sp,$sp,48
.LCFI0:
        sd      $fp,40($sp)
```

```
.LCFI1:
        sd      $28,32($sp)
.LCFI2:
        move    $fp,$sp
.LCFI3:
        .set    noat
        lui     $1,%hi(%neg(%gp_rel(count)))
        addiu   $1,$1,%lo(%neg(%gp_rel(count)))
        daddu   $gp,$1,$25
        .set    at
        sw      $4,16($fp)
        sw      $0,24($fp)
        li      $2,1                    # 0x1
        sw      $2,20($fp)
.L3:
        lw      $2,20($fp)
        lw      $3,16($fp)
slt     $2,$3,$2
        beq     $2,$0,.L6
        b       .L4
L6:
        lw      $2,24($fp)
        lw      $3,20($fp)
        addu    $2,$2,$3
        sw      $2,24($fp)
.L5:
        lw      $2,20($fp)
        addu    $3,$2,1
        sw      $3,20($fp)
        b       .L3
.L4:
        lw      $3,24($fp)
        move    $2,$3
        b       .L2
.L2:
        move    $sp,$fp
        ld      $fp,40($sp)
        ld      $28,32($sp)
        addu    $sp,$sp,48
        j       $31
.LFE1:
        .end    count
```

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

MIPS Instruction Set

Introduction

- Objective: To introduce the MIPS instruction set and to show how MIPS instructions are represented in the computer.

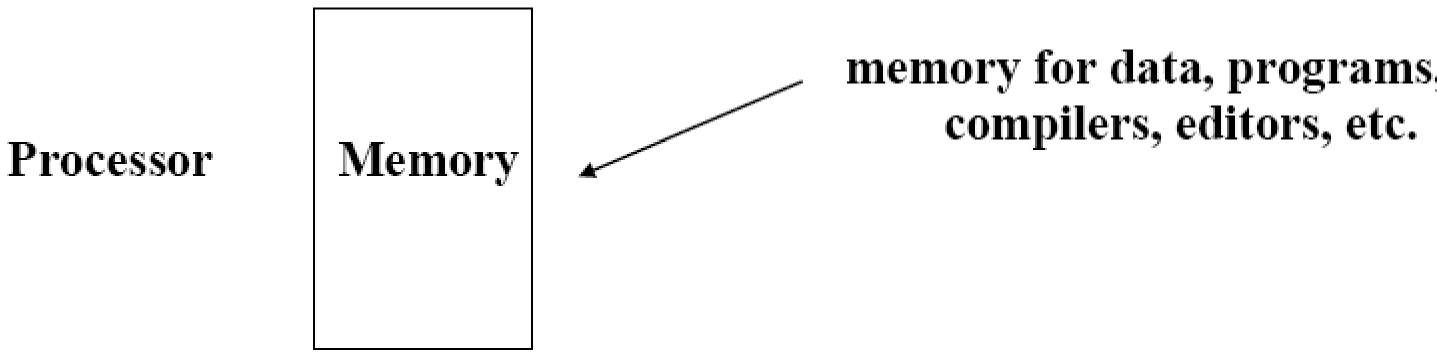
- The stored-program concept:

  - Instructions are represented as numbers.

  - Programs can be stored in memory to be read or written just like data.

Stored Program Concept

- Instructions are just a sequence of 32 bits
- Programs are stored in memory
  — to be read or written just like data



- Fetch & Execute Cycle
  - Instruction is fetched and put into a special register
  - Bits in the instruction register determine the subsequent actions
  - When done, fetch the next instruction and continue execution

MIPS Arithmetic

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first)

- Example

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

- Using the natural number of operands for an operation (e.g. addition) conforms to the design principle of keeping the hardware simple.

Temporary Variables

- Regularity of instruction format requires that expressions get mapped to a sequence of binary operations with temporary results being stored in temporary variables.

- Example

C code:

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

Assume f, g, h, i, j are in $s0 through $s4 respectively

MIPS code:

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

Registers vs. Memory

- Operands for arithmetic instructions must be registers,
  — only 32 integer registers are available

- Compiler associates variables with registers

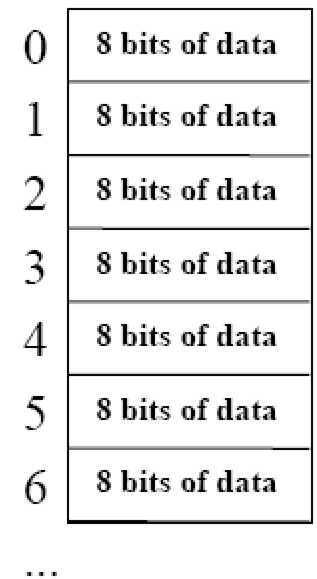
- When too many variable are used to fit in 32 registers, the compiler must allocate temporary space in memory and then load and store temporary results to/from memory.
- The compiler tries to put most frequently occurring variables in registers.
- The extra temporary variables must be “spilled” to memory.

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Memory Organization

- Viewed as a large, single-dimension array, where a memory address is an index into the array
- MIPS systems address memory in byte chunks
- The memory address (= index) points to a byte in memory.
- This is called "Byte addressing”



Memory Organization

- Most data items are grouped into words
- A MIPS word is 4 bytes or 32 bits

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

Registers also hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned (alignment restriction)
- Bytes can be accessed from left to right (big endian) or right to left (little endian).

Load and Store

- All arithmetic instructions operate on registers
- Memory is accessed through load and store instructions

- An example C code: `A[12] = h + A[8];`

  Assume that $s3 contains the base address of A

  MIPS code:

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

- Note: sw (store word instruction) has destination last.
- Note: remember arithmetic operands are registers, not memory!

  This is invalid: `add 48($s3), $s2, 32($s3)`

Logical Operations

- **Instructions for bitwise manipulation**

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

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

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **Instruction** / **Meaning**

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Our First Example

- **Can we figure out the code?**

```
swap(int v[], int k)
{ int temp;
temp = v[k]
v[k] = v[k+1];
v[k+1] = temp;
}
```

```
swap:
sll $2, $5, 2
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

```
swap(int v[], int k)
      {
             int temp;
             temp = v[k]
             v[k] = v[k+1];
             v[k+1] = temp;
      }
```

$5 = k

$2 = 4k

$4 = ADDR of v

$2 = $5*4

```
swap:

      sll $2, $5, 2

      add $2, $4, $2

      lw $15, 0($2)

      lw $16, 4($2)

      sw $16, 0($2)

      sw $15, 4($2)

      jr $31
```

$2 = $4 + 4k ADDR OF v[k]

TEMP = v[k] $15 = TEMP

$16 = v[k+1]
v[k] = $16 v[k] = v[k+1]

v[k+1] = TEMP

Machine Language

- Instructions, registers and data words are 32 bit long
  - Example: `add $t1, $s1, $s2`
  - Registers have numbers/indeces: `$t1=9, $s1=17, $s2=18`
- Instruction Format:

000000	10001	10010	01001	00000	100000

op	rs	rt	rd	shamt	funct

- Guess what do the field names stand for?

Register Names

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

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **shamt: how many positions to shift**
- **Shift left logical**
  - **Shift left and fill with 0 bits**
  - **`sll` by *i* bits multiplies by 2^i**
- **Shift right logical**
  - **Shift right and fill with 0 bits**
  - **`srl` by *i* bits divides by 2^i (unsigned only)**

AND Operations

- **Useful to mask bits in a word**
  - **Select some bits, clear others to 0**

```
and $t0, $t1, $t2
```

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

OR Operations

- **Useful to include bits in a word**
  - **Set some bits to 1, leave others unchanged**

```
or $t0, $t1, $t2
```

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- **Useful to invert bits in a word**
  - **Change 0 to 1, and 1 to 0**
- **MIPS has NOR 3-operand instruction**
  - **a NOR b == NOT (a OR b)**

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-format type for data transfer instructions
  - The other format was R-type for register (add and sub)
- Example: `lw $t0, 32($s2)`

35	18	8	32

op	rs	rt	16 bit number

- Where's the compromise?

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

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

Immediate Instructions

Immediate mode includes small constants in instruction

Avoid extra memory operations

Example: addi $s1, $s1, 1

8	17	17	1

Addressing in Jumps

- **J format format (jump format – j, jal)**

op	address
6-bits	26-bits

- **Example: j 10000**

2	10000
6-bits	26-bits

Target Addressing Example

- **Loop code example**
  - **Assume Loop at location 80000**

`Loop:`	`sll  $t1, $s3, 2`	80000	0	0	19	9	2	0
	`add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
	`lw   $t0, 0($t1)`	80008	35	9	8	0		
	`bne  $t0, $s5, Exit`	80012	5	8	21	2		
	`addi $s3, $s3, 1`	80016	8	19	19	1		
	`j    Loop`	80020	2	20000				
`Exit:`	`…`	80024						

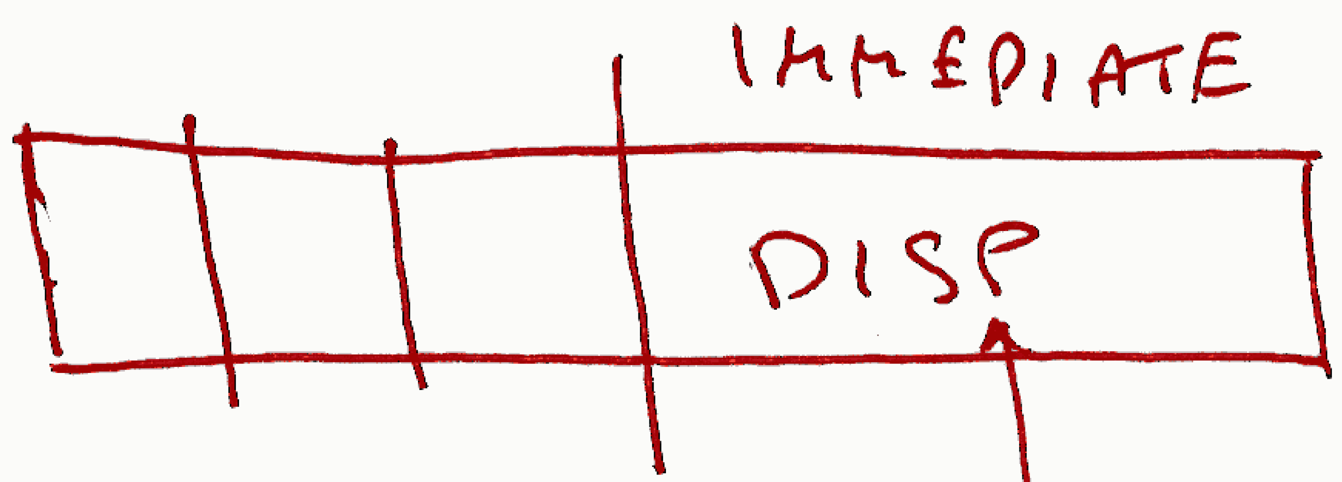
OPCODE	TARGET
2	

ENTER TARGET/4

80000/4 = 20000

Loop:	sll	$t1, $s3, 2	80000	0	0	19	9	2	0
	add	$t1, $t1, $s6	80004	0	9	22	9	0	32
	lw	$t0, 0($t1)	80008	35	9	8	0		
	bne	$t0, $s5, Exit	80012	5	8	21	2		
	addi	$s3, $s3, 1	80016	8	19	19	1		
	j	Loop	80020	2	20000				
Exit:	…		80024						

OPCODE



$$\text{DEST} = \text{CUR} + 4 + \text{DISP} * 4$$

$$80024 = 80012 + 4 + \text{DISP} * 4$$

$$8 = \text{DISP} * 4$$

$$\text{DISP} = 2$$

Loop:	sll $t1, $s3, 2	80000	0	0	19	9	2	0
	add $t1, $t1, $s6	80004	0	9	22	9	0	32
	lw $t0, 0($t1)	80008	35	9	8	0		
	bne $t0, $s5, Exit	80012	5	8	21	2		
	RT RS			RS	RT			
	addi $s3, $s3, 1	80016	8	19	19	1		
	j Loop	80020	2	20000				
Exit:	...	80024						

CURRENT ADDRESS

DESTINATION ADDRESS

No-Op Instructions

- **What would you expect a no-op instruction to be in binary?**
- **What is this in assembly?**

Design Principles

- Simplicity favors regularity
  - All instructions 32 bits
  - All instructions have 3 operands
- Smaller is faster
  - Only 32 registers
- Good design demands good compromises
  - All instructions are the same length
  - Limited number of instruction formats: R, I, J
- Make common cases fast
  - 16-bit immediate constant
  - Only two branch instructions

Branching and Procedures in MIPS

Introduction

- **Objective: To illustrate how programming constructs such as conditionals, loops and procedures can be translated into MIPS instructions.**

Conditional Operations

- **Branch to a labeled instruction if a condition is true**
  - **Otherwise, continue sequentially**
- **`beq rs, rt, L1`**
  - **if (rs == rt) branch to instruction labeled L1;**
- **`bne rs, rt, L1`**
  - **if (rs != rt) branch to instruction labeled L1;**
- **`j L1`**
  - **unconditional jump to instruction labeled L1**

Control

- **Decision making instructions**
  - **Alter the control flow – change the "next" instruction to be executed**

- **MIPS conditional branch instructions:**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- **Example:** `if (i==j) h = i + j;`

```
    bne $s0, $s1, Label
    add $s3, $s0, $s1
Label:    ....
```

If-Then Structure

- MIPS unconditional branch instructions:
  `j label`
- Example:

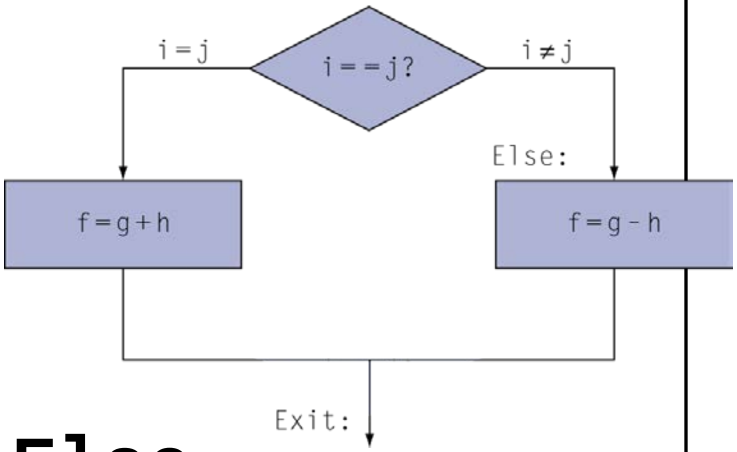
```
if (i!=j)              beq $s4, $s5, Equal
    h=i+j;             add $s3, $s4, $s5
else                   j GoOn
    h=i-j;             Equal: sub $s3, $s4, $s5
                       GoOn: ...
```

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

Compiling Loop Statements

- **C code:**

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

  - **i in $s3, k in $s5, address of save in $s6**
- **Compiled MIPS code:**

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

Branch Instruction Design

- Why not `blt`, `bge`, etc?

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Less Than Test

- What about Branch-if-less-than?
- New instruction:

```
                          if  $s1 < $s2 then
                                 $t0 = 1
    slt $t0, $s1, $s2     else
                                 $t0 = 0
```

- MIPS does not include blt, bgt, ble, bge, bgz, etc. instructions because it is considered too complicated.
- Assembler pseudoinstruction: "`blt $s1, $s2, Label`"
- Note that the assembler needs a register to do this, — there are important conventions for registers use

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

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

Quick review

- **Instruction** | **Meaning**

Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,Label`	`Jump to Label if $s4≠$s5`
`beq $s4,$s5,Label`	`Jump to Label if $s4=$s5`
`j Label`	`Next instr. is at Label`
`slt $t0, $s1, $s2`	`Set $t0 = 1 if $s1 < $s2` `else set $t0 = 0`

Quick review

- **Formats**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Loops

- Compiling a while loop (Assume i and k correspond to $s3, and $s5 and that the base address of *save* is in $s6.)

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

```
Loop: sll $t1, $s3, 2     # $t1 = 4 * i
      add $t1, $t1, $s6   # address of save[i]
      lw $t0, 0($t1)      # get save[i]
      bne $t0, $s5, Exit  # goto Exit if save[i] ≠ k
      add $s3, $s3, 1     # i = i + 1
      j Loop
Exit:
```

Procedures

- **In the execution of a procedure, the program must follow these steps:**
  - **Place parameters in a place where the procedure can access them**
  - **Transfer control to the procedure**
  - **Acquire the storage resources needed for the procedure**
  - **Perform the desired task**
  - **Place the result where the calling program can access it**
  - **Return control to the point of origin**

Registers for Procedure Calling and the jal Instruction

- $a0 - $a3: four argument registers used to pass parameters
- $v0 - $v1: two value registers in which to return values
- $ra: one return address register to return to the point of origin
- jal ProcedureAddress: instruction to transfer control to a procedure and store the return address in $ra ($ra is set to PC + 4, address of the next instruction after procedure call)
- jr $ra - used to transfer control back to the calling program

Saving Registers using a Stack

- Additional registers used by the called procedure must be saved prior to use, or the values used by the calling procedure will be corrupted.
- The old values can be saved on a stack (call stack). After the called procedure completes, the old values can be popped off the stack and restored.
- $sp: stack pointer register contains the address of the top of the stack. By convention, address on the stack grows from higher addresses to lower address, which implies that a push subtracts from $sp and a pop adds to $sp.

Policy of Use Conventions

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Register 1 ($at) reserved for assembler, 26-27 for the operating system.

Register Conventions

- The 8 "saved" registers \$s0 - \$s7 must be preserved on a procedure call, i.e. the called procedure must save these before using them.

- The 10 "temporary" registers \$t0 - \$t9 are not preserved by the called procedure. The calling procedure can not assume they will not change after a procedure call and, hence, must save them prior to the call if the values are needed after the call.

- Saved registers should be used for long lived variables, while temporary registers should be used for short lived variables

Nested Procedures and Automatic Variables

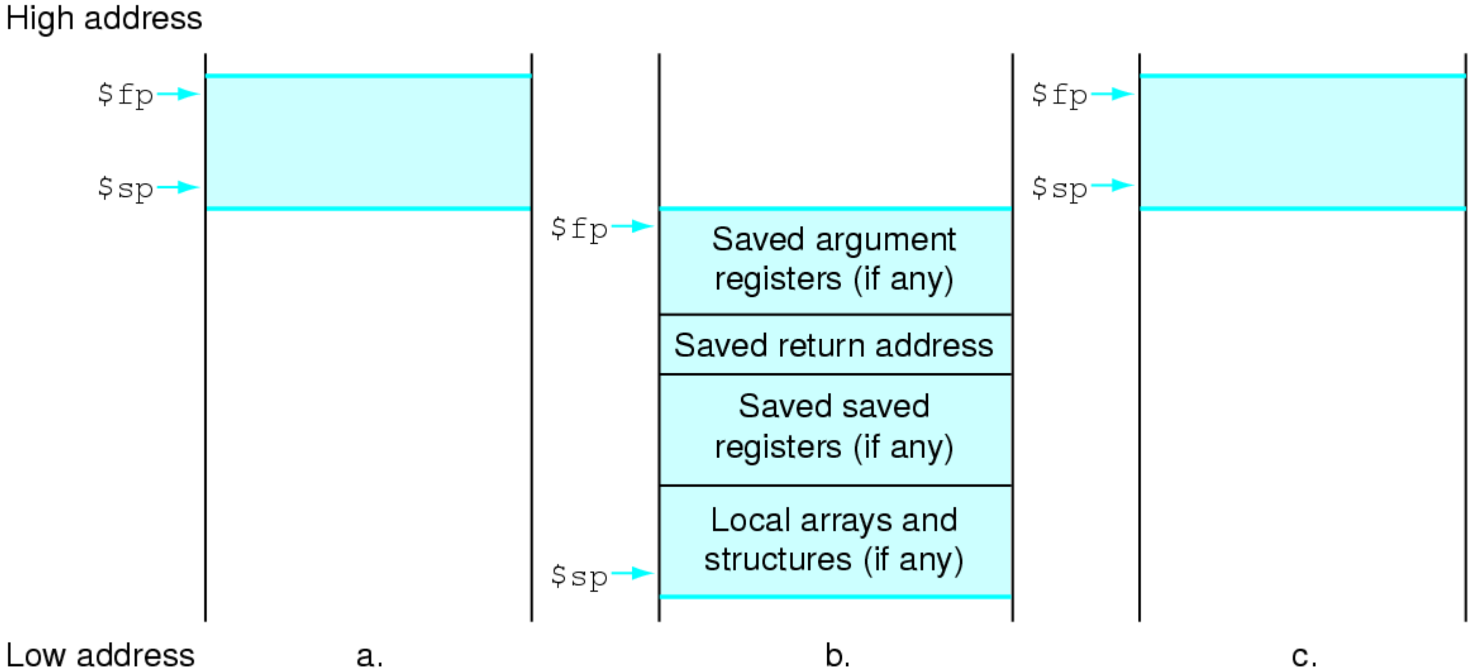
- **A new call frame or activation record must be created for each nested procedure call.**

- **Argument registers and the return address register must be saved in addition to saved registers since new values will be put in them for the nested procedure call.**

- **Automatic variables (i.e. variables that are local to a procedure and are discarded when the procedure completes) are also allocated on the call stack. They are popped when the call completes.**

Procedure Activation Records (Frames) and the Call Stack

- **An activation record (frame) is a segment on the stack containing a procedure's saved registers and local variables.**
- **Each time a procedure is called a frame ($fp: frame pointer register points to the current frame) is placed on the stack.**



Leaf Procedure

```
/* Example from page 134 */

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

Leaf Procedure

```
func:   sub   $sp,$sp,4        # push stack and save registers
        sw    $s0,0($sp)

        add   $t0,$a0,$a1      # g + h
        add   $t1,$a2,$a3      # i + j
        sub   $s0,$t0,$t1      # (g+h) - (i+j)
        add   $v0,$s0,$zero    # return f = (g+h)-(i+j)

        lw    $s0,0($sp)       # restore registers and pop stack
        add   $sp,$sp,4
        jr    $ra              # return to calling program
```

MUST BE SAVED BEFORE USE

func (g, h, i, j)

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

ret (f)

fix THE STACK POINTER

Leaf Procedure

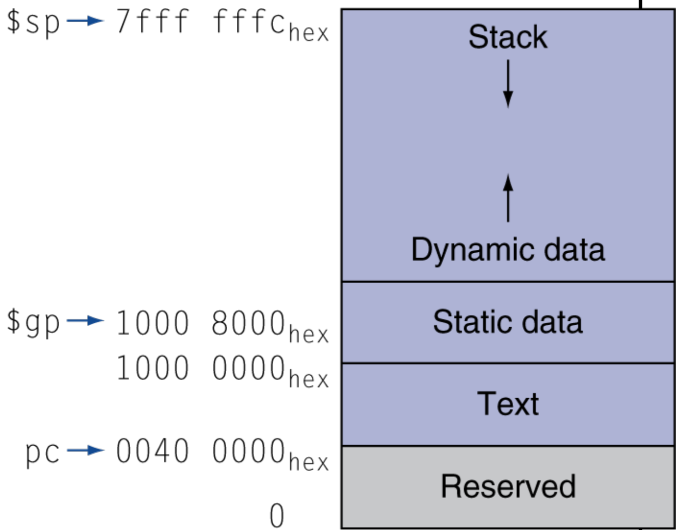
```
func:   sub   $sp,$sp,4          # push stack and save registers
        sw    $s0,0($sp)

        add   $t0,$a0,$a1        # g + h
        add   $t1,$a2,$a3        # i + j
        sub   $s0,$t0,$t1        # (g+h) - (i+j)
        add   $v0,$s0,$zero      # return f = (g+h)-(i+j)

        lw    $s0,0($sp)         # restore registers and pop stack
        add   $sp,$sp,4
        jr    $ra                # return to calling program
```

Memory Layout

- **Text: program code**
- **Static data: global variables**
  - **e.g., static variables in C, constant arrays and strings**
  - **$gp initialized to address allowing ±offsets into this segment**
- **Dynamic data: heap**
  - **E.g., malloc in C, new in Java**
- **Stack: automatic storage**



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

- **C code (naïve):**
  - **Null-terminated string**

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

  - **Addresses of x, y in $a0, $a1**
  - **i in $s0**

String Copy Example

- **MIPS code:**

```
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
```

Recursive Procedure

```
/* Factorial example from pp. 136-137 */

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

```
fact:
        sub    $sp,$sp,8        # push stack
        sw     $ra,4($sp)       # save return address
        sw     $a0,0($sp)       # save n

        slt    $t0,$a0,1        # test n < 1
        beq    $t0,$zero,L1     # branch if n >= 1
        add    $v0,$zero,1      # return 1
        add    $sp,$sp,8        # pop stack
        jr     $ra              # return to calling procedure
L1:
        sub    $a0,$a0,1        # set parameter to n-1
        jal    fact             # call fact(n-1)
        lw     $a0,0($sp)       # restore previous value of n
        lw     $ra,4($sp)       # restore previous return address
        mul    $v0,$a0,$v0      # return n * fact(n-1)

        add  $sp,$sp,8          # pop stack
        jr   $ra                # return to calling procedure
```

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing and Array

<table>
<tr>
<td>
<pre>
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
</pre>
</td>
<td>
<pre>
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
        p = p + 1)
    *p = 0;
}
</pre>
</td>
</tr>
<tr>
<td>
<pre>
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2   # $t1 = i * 4
       add $t2,$a0,$t1 # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1 # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
</pre>
</td>
<td>
<pre>
       move $t0,$a0   # p = & array[0]
       sll $t1,$a1,2   # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
</pre>
</td>
</tr>
</table>

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer