Part II
Introduction to MIPS Instruction Set Architecture

Overview

This section looks at details of MIPS programming. It talks about subroutines, branches and registers – lots of different paving stones on our road to knowledge about MIPS.

- **C/Assembly Decisions – Section 2.6:**
  - **`if`**, `if-else`
- **Inequalities**
- **C/Assembly Loops:**
  - `while(){}` , `do {} while`, `for() {}`
- **C Switch Statement**
- **Stack – Section 2.7**
- **Procedures – Section 2.7**

So Far...

- All instructions have allowed us to manipulate data.
- So we've built a calculator that lets us add and subtract.
- To build a computer, we need ability to make decisions

C Decisions: `if` Statements

- **2 kinds of `if` statements in C**
  - `if (`*condition*`)` *clause*
  - `if (`*condition*`)` *clause1* `else` *clause2*
- **Rearrange 2nd `if` into following:**

```
if  (condition) goto L1;
      clause2;      # Do the work of the else
      go to L2;

L1: clause1;

L2:  # Continue on
```

  - Not as elegant as if - else, but same meaning

MIPS Decision Instructions

Conditional Branch

- **Decision instruction in MIPS:**
  - `beq      register1, register2, Label`
  - `beq` is 'Branch if (registers are) equal'. Same meaning as (using C):
    `if      (register1==register2) goto Label`
- **Complementary MIPS decision instruction**
  - `bne      register1, register2, Label`
  - `bne` is 'Branch if (registers are) not equal' Same meaning as (using C):
    `if      (register1!=register2) goto Label`
  - Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition

Unconditional Branch

- **Same meaning as (using C):**

`goto label`

- **Technically, the same as:**

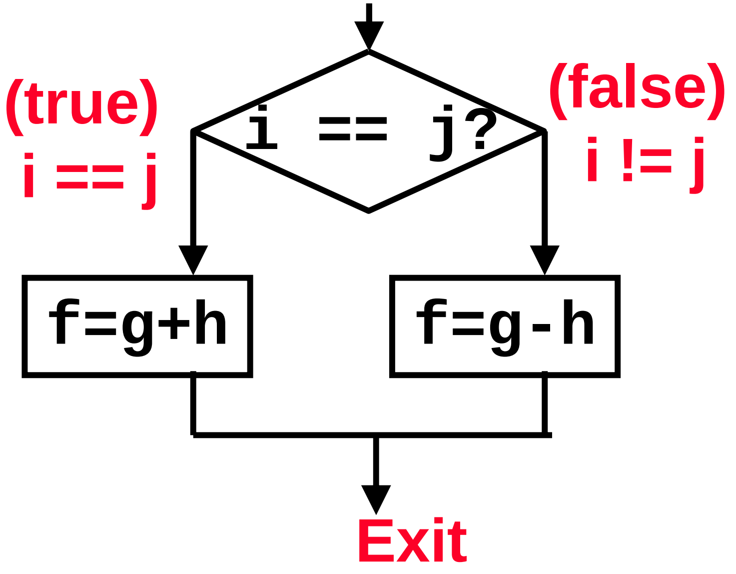
`j label` → **`beq   $0, $0, label`**

since it always satisfies the condition.

Compiling C `if` into MIPS

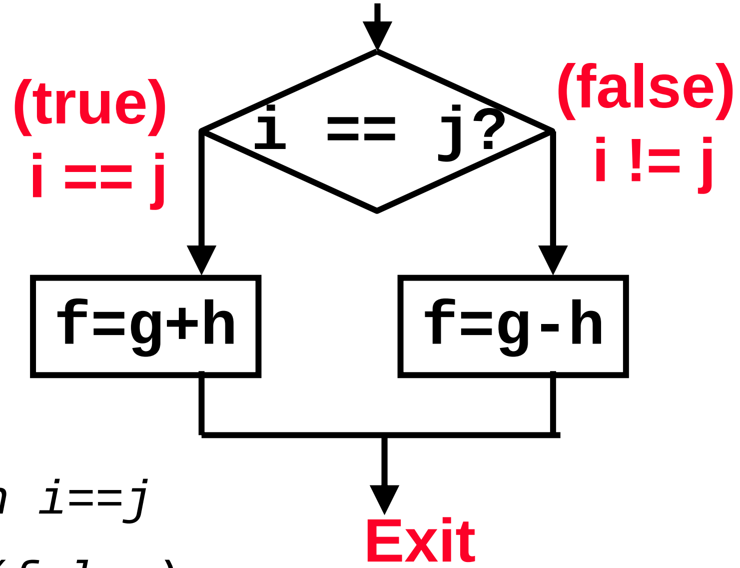
- **Compile by hand**

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



- **Use this mapping:**
  - `f`: `$s0`,
  - `g`: `$s1`,
  - `h`: `$s2`,
  - `i`: `$s3`,
  - `j`: `$s4`

Compiling C `if` into MIPS



- **Final compiled MIPS code:**

```
        beq   $s3, $s4, True     # branch i==j
        sub   $s0, $s1, $s2      # f=g-h(false)
        j       Fin              # go to Fin
    True:
        add    $s0,$s1,$s2       # f=g+h (true)
    Fin:
```

- **Note: Compilers automatically create labels to handle decisions (branches) appropriately. Generally not found in HLL code.**

Inequalities in MIPS

- **Until now, we've only tested equalities (== and != in C). General programs need to test < and > as well.**

- **Create a MIPS Inequality Instruction:**
  - <Set on Less Than>
  - Syntax: `slt  reg1,reg2,reg3`
  - Meaning:

```
if (reg2 < reg3)
     reg1 = 1;
 else

     reg1 = 0;
```

  - In computereeze, "set" means "set to 1", "reset" or "clear" means "set to 0".

- **Compile by hand:**

```
if (g < h) goto Less;
```

- **Use this mapping:**

  g: $s0, h: $s1

Inequalities in MIPS

- **Final compiled MIPS code:**

```
slt $t0,$s0,$s1      # $t0 = 1 if g<h

bne $t0,$0,Less      # goto Less

                     # if $t0!=0
                     # (if (g<h))   Less:
```

- **Branch if `$t0` != 0 or (g < h)**
  - Register $0 always contains the value 0, so bne and beq often use it for comparison after an `slt` instruction.

Inequalities in MIPS

- 4 combinations of slt & beq / bneq:

```
slt $t0,$s0,$s1    # $t0 = 1 if g<h
  bne $t0,$0,Less    # if(g<h) goto Less

 slt $t0,$s1,$s0  # $t0 = 1 if g>h
  bne $t0,$0,Grtr # if(g>h) goto Grtr

slt $t0,$s0,$s1    # $t0 = 1 if g<h
  beq $t0,$0,Gteq    # if(g>=h) goto Gteq

slt $t0,$s1,$s0    # $t0 = 1 if g>h
  beq $t0,$0,Lteq    # if(g<=h) goto Lteq
```

Immediates in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`
  - Helpful in `for` loops

C

```
if (g >= 1) goto Loop

 Loop:   . . .
```

MIPS

```
slti $t0,$s0,1     # $t0 = 1 if
                   # $s0<1 (g<1)
beq  $t0,$0,Loop   # goto Loop

                   # if $t0==0
                   # (if (g>=1))
```

Loops in C/Assembly

- **There are three types of loops in C:**
  - `while`
  - `Do while`
  - `for`

- **Each can be rewritten as either of the other two, so the method used in the previous example can be applied to `while` and `for` loops as well.**

- **Key Concept: Though there are multiple ways of writing a loop in MIPS, conditional branch is key to decision making**

Example: The C Switch Statement

- **Choose among four alternatives depending on whether k has the value 0, 1, 2 or 3. Compile this C code:**

```
switch (k) {
    case 0: f=i+j; break; /* k=0*/
    case 1: f=g+h; break; /* k=1*/
    case 2: f=g-h; break; /* k=2*/
    case 3: f=i-j; break; /* k=3*/
    }
```

- **This is complicated, so simplify.**

- **Rewrite as a chain of if-else statements - we already know how to do this:**

```
if(k==0) f= I + j;
  else if(k==1) f= g + h;
    else if(k==2) f= g - h;
      else if(k==3) f= - j;
```

- **Use this mapping:**

  f: \$s0, g: \$s1, h: \$s2, i: \$s3, j: \$s4, k: \$s5

Example: The C Switch Statement

- **Final compiled MIPS code:**

```
     bne    $s5, $0, L1         # branch k!=0
     add    $s0, $s3, $s4       # k==0 so f=i+j
     j      Exit                # end of case so Exit
L1:
    addi    $t0, $s5, -1        # $t0 = k-1
    bne     $t0, $0, L2         # branch k != 1
    add     $s0, $s1, $s2       # k==1 so f=g+h
    j       Exit                # end of case so Exit
L2:
    addi    $t0, $s5, -2        # $t0=k-2
    bne     $t0, $0, L3         # branch k != 2
    sub     $s0, $s1, $s2       # k==2 so f=g-h
    j       Exit                # end of case so Exit
L3:
    addi    $t0, $s5, -3        # $t0 = k-3
    bne     $t0, $0, Exit       # branch k != 3
    sub     $s0, $s3, $s4       # k==3 so f=i-j
Exit:
```

Instruction Support for Functions

C

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

M I P S

```
address
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 ...

2000 sum: add $v0,$a0,$a1
2004 jr   $ra            # new instruction
```

Support for Functions – jal & jr

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

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

- **After:**

```
1012 jal sum  # $ra=1016,go to sum
```

- **Why have a `jal`? Make the common case fast: functions are very common.**
- **Syntax for `jr` (jump register):**

```
jr register
```

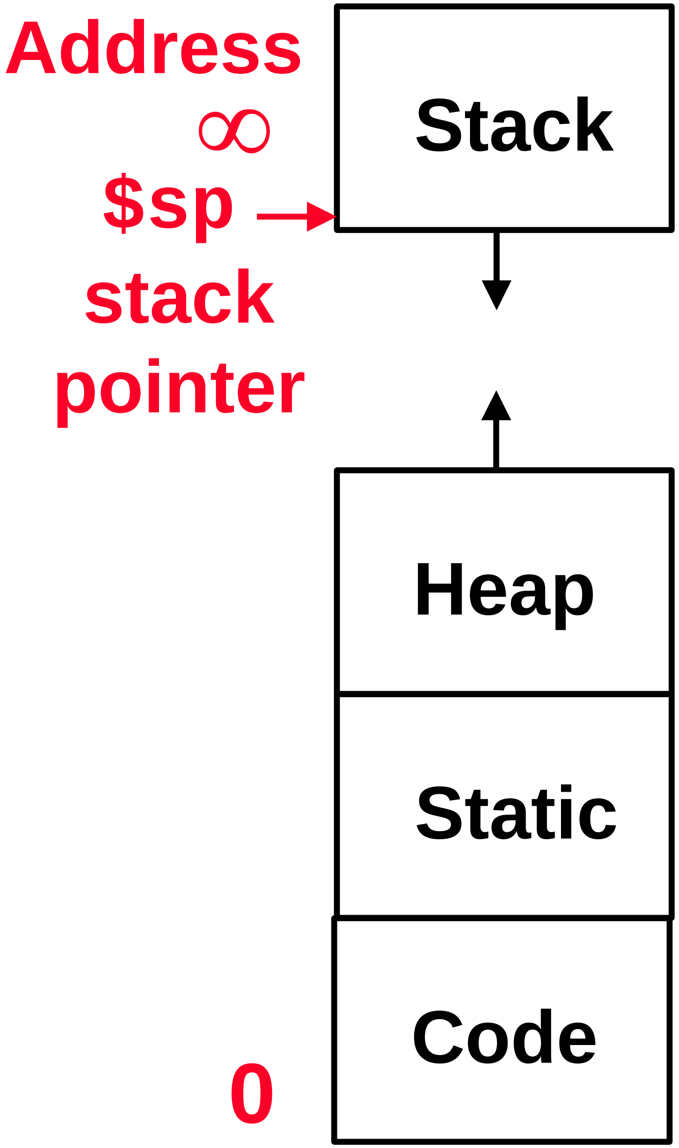
- **Instead of providing a label to jump to, the `jr` instruction provides a register which contains an address to jump to.**
- **Very useful for function calls:**
  - `jal` stores return address in register ($ra)
  - `jr` jumps back to that address

Nested Procedures – Why have a stack

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

- **Routine 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`.**
- **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
  - Heap: Variables declared dynamically
  - Stack: Space to be used by procedure during execution; this is where we can save register values

C memory Allocation



Space for saved procedure information

Explicitly created space, *e.g.*, malloc(); C pointers

Variables declared once per program

Executable Program

Using the Stack

- 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)

- Compile by hand

```
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
```

Steps for Making a Procedure Call

1) Save necessary values onto stack.

2) Assign argument(s), if any.

3) `jal` call

4) Restore values from stack.

- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in $a0, $a1, $a2 and $a3
- Return value is always in $v0 (and if necessary in $v1)
- Must follow register conventions (even in functions that only you will call)! So what are they?

Example: Compile This

```
main() {
    int i,j,k,m; /* i-m:$s0-$s3 */
   i = mult(j,k); ... ;
    m = mult(i,i); ...
}
int mult (int mcand, int mlier){
    int product;
   product = 0;
    while (mlier > 0)  {
       product += mcand;
       mlier -= 1;
   }
    return product;
}
```

Example: Compile This

```
main:
 addi $sp, $sp, -4

 sw   $ra,0($sp)

 add  $a0, $s1, $0          # arg0 = j

 add  $a1, $s2, $0          # arg1 = k

 jal  mult                  # call mult

                            # result is in $v0 on return

 add  $a0, $v0, $0          # arg0 = i
 add  $a1, $s0, $0          # arg1 = i
 jal  mult                  # call mult
                            # Pass result back in $v0
 lw   $ra,0($sp)
 addi $sp,$sp,8
 jr   $ra
```

Example: Compile This

```
mult:
      add  $t0,$0,$0        # prod = 0

Loop:
      slt  $t1,$0,$a1       # mlr > 0?
      beq  $t1,$0,Fin       # no => Fin
      add  $t0,$t0,$a0      # prod += mc
      addi $a1,$a1,-1       # mlr -= 1
      j     Loop            # goto Loop

Fin:
      add  $v0,$t0,$0       # $v0 = prod
      jr   $ra              # return
```

Example: Compile This

- **Notes:**
  - no `jal` calls are made from `mult` and we don't use any saved registers, so we don't need to save anything onto stack
  - temp registers are used for intermediate calculations (could have used s registers, but would have to save the caller's on the stack.)
  - $a1 is modified directly (instead of copying into a temp register) since we are free to change it
  - result is put into $v0 before returning

Things to Remember

- **A Decision allows us to decide which pieces of code to execute at run-time rather than at compile-time.**
- **C Decisions are made using conditional statements within an `if`, `while`, `do while` or `for`.**
- **MIPS Decision making instructions are the conditional branches: beq and bne.**
- **To help the conditional branches make decisions concerning inequalities, we introduce a single instruction: <Set on Less Than> called `slt`, `slti`, `sltu`, `sltui`**