Label Naming

Labels and Scope

- make *labels* readable, descriptive and intuitive, just like variable and function names
- *labels* must be unique within scope
- assume they only need to be unique within a single source file for now (i.e. you can use same *label* in different files)
- later on you will learn how deal with *labels* that must be understood by other files (i.e. externally/globally)
- *labels* can be generated manually vs. automatically

Example: if…else

Implementing if ... else ...

In C++

```
if (r1 == 0)
    r2 = r2 + GST; //then part, add GST
else
    r2 = r2 + HST; // else part, add HST
```

assume GST is in r3 and HST is in r4

```
        beq $1, $0, useGST   ; if r1==0
        add $2, $2, $4       ; else part
        beq $0, $0, final    ; always equal
useGST: add $2, $2, $3       ; then part
final:  ; continue with program
```

Assembly File

What does an Assembly File Contain?

- assembly instructions
- label declarations
- data definitions (.word)
- comments – start with semicolon
- numbers can be: hexadecimal, positive or negative decimal
  - hexadecimal: use 0x prefix, e.g. 0x20
  - positive decimal: don’t use 0x prefix, e.g. 32
  - negative decimal: don’t use 0x prefix, but do use a negative sign e.g. -32

Assembly File

Suggested Format

- Three columns
  1. label
  2. instruction / data
  3. comment
- e.g. load a hexadecimal, a positive decimal , a negative decimal

Labels *Instructions/Data* *Comments*

```
loop:    lis $1
         .word 0x20      ; $1 = 32 in decimal
         lis $2
         .word 32        ; $2 = 32
         lis $3
         .word -32       ; $2 = -32
```

Assembly File

Example: Absolute Value

- Task: Compute the absolute value of $1, store it in $1, then return.
- *in C / C++*

```
if (r1 < 0) {r1 = 0 - r1; } return;
```

- *in MIPS assembly language*

Labels *Instructions/Data* *Comments*

```
        slt $2,$1,$0        ; is $1 < 0 ?
        beq $2,$0,end       ; if false, goto end
        sub $1,$0,$1        ; else negate $1
end:    jr $31;
```

Assembly File

Example: Sum Integers in C

- Task: Sum the integers 1..13, store in r3, then return.
- *In C / C++*…

```
int r2 = 13; // integers to be summed
int r3 = 0;  // answer
int r1 = 1;  // reduce r2 by 1 each time

while (r2 != 0) {
  r3 = r3 + r2;  // r3 = 13 + 12 + 11 + …
  r2 = r2 - r1;  // r2 = 13, 12, 11, …
}
return;
```

Assembly File

Example: Sum Integers in MIPS Assembly Language

- Task: Sum the integers 1..13, store in $3, then return.

Labels *Instructions/Data* *Comments*

```
        lis $2              ; $2 = 13
        .word 13
        add $3,$0,$0        ; $3 = 0
        lis $1              ; $1 = 1
        .word 1
loop:   add $3,$3,$2        ; $3 = $3 + $2
        sub $2,$2,$1        ; $2 = $2 - 1
        bne $2,$0,loop      ; loop until $2==0
        jr $31              ; return
```

Arrays

Indexing into an Array

- You have an array, A, where
  - the indices start at 0, i.e. A[0], A[1], A[2], …
  - the size of each element in the array is *4* bytes.
- If the address of A[0] is in register $1, then
  - the address of A[1] is *4* ($1),
  - the address of A[2] is *8* ($1),

    …
  - the address of A[*i*] is 4*i* ($1)
- The address of the 0^{th} element is called the *base address.*
- *The address of the i^{th} element is*

  base address + (i × size of an element)

Arrays

Example: Accessing the 5th element of an array

$1 base address of array $3 the 5th element

$4 the size of each element $5 temp storage

```
lis $5                ; Get 5th element
.word 5
lis $4                ; Size of each element
.word 4               ;
mult $5,$4            ; Calc and store offset
mflo $5               ;  of element 5 in $5
add $5,$1,$5          ; Address of element 5
lw $3,0($5)           ; Load A[5] into $3
jr $31                ; Return
```

Output

Memory Mapped I/O

- input /output from devices (such as a keyboard, mouse or screen) are treated just like reading from and writing to memory
- i.e. use MIPS instructions **`lw`** and **`sw`**, but to specific memory locations
- For CS 241, to output a char to the screen, store the ASCII value of the character to memory address FFFF000C_{hex}
  - also called video output
  - bytes written to this address appear on screen
  - write one character at a time

Output Example

Memory Mapped I/O

```
;; Print "CS\n" on the screen
    lis $1              ; address of output buffer
   .word 0xFFFF000C
    lis $2
    .word 67            ; ASCII C
    sw $2,0($1)         ; write to screen
    lis $2
    .word 83            ; ASCII S
    sw $2,0($1)         ; write to screen
    lis $2
    .word 10            ; ASCII newline
    sw $2,0($1)         ; write to screen
    jr $31              ; return
```

Subroutines

Key Challenges in Implementing Subroutines

- How do we ensure that essential data stored in registers is not lost?
  - hint: store them in RAM
- How do we call and return from a subroutine?
  - hint: `jalr` and `jr`
- How do we pass parameters to the subroutine?
  - hint: on the call stack
- How do we return values from a subroutine?
  - hint: agree an a register

Subroutines

Subroutines vs. Functions

- *subroutines*: assembly language's version of functions
- programmers must do more work, essentially implement a function using: labels, PC, `lw`, `sw`
- *function name* ⇒ go to this label / memory location and start executing the instructions you find there
- *arguments and return values* ⇒ agree to place certain values in certain registers or memory locations
  - *gone*: no concept of type checking
- *local scope, variables* ⇒ *gone*: can access any register and most memory locations (more on that later)

Subroutines

```
if {amount_requested > account_balance)
   printf("Request a lower amount")
else {
   printf("Collect money from dispenser")
   dispense(amount_requested)
}
```

Challenges of Using Subroutines

- call/return – how to redirect execution?
  - call is *static* ⇒ always go to same location
    e.g. the beginning of the printf function
  - return is *dynamic* ⇒ must track where to return to
    e.g. which line of C++ called the printf function
- complications: nested call/return, recursion

Subroutines

Two Instructions

`jalr $s`

- meaning: *jump and link register*
- copy the address of next instruction (PC) to $31
- set PC to the address stored in `$s`
- start executing code at this new location

`jr $s`

- meaning: *jump (to the address in) register $s*
- set PC to $s
- start executing code at this new location
- convention: register $31 holds return address

Subroutines

Storing Essential Data

- A subroutine can call another subroutine (or itself)
- What about registers that are in use?
- For example, say we have
  - important data stored in registers 1 to 4
  - want to call subroutine *func* which uses registers 2 and 3 as "local variables" / temporary values
  - registers ≠ local variables, i.e. subroutine *func* will overwrite these important values
- must save *current execution context* (set of register values) before jumping to *func* and restore the context once *func* has finished
- *Key Question:* save where?

The Run-time Stack

Solution: Use a stack

- a.k.a. the *call stack* or the *run-time stack*
- use part of memory (i.e. RAM) as a stack
  - last-in first-out queue
- *convention:* stack grows downward in memory
- *convention:* the address of the bottom of the stack stored in the stack pointer (SP) register
- *convention:* typically register $29 is the SP in MIPS
- *exception:* in our MIPS simulator we use $30

The Run-time Stack

Saving and Restoring Context on the Stack

- *save* (a.k.a.) *push onto the stack*
- two step process
  1. store the register values
  2. decrement stack pointer (SP) to reflect the change
- e.g. say we want to store the values in $2 and $3

```
func:  sw $2,-4($30)       ; 1. Store register
       sw $3,-8($30)       ;    values on stack
       lis $3              ; 2. Decrement SP
       .word 8             ;    by 8.
       sub $30,$30,$3
       ...                 ; body of function
```

The Run-time Stack

Saving and Restoring Context on the Stack

- *restore* (a.k.a.) *pop off the stack*
- two step process
  1. increment stack pointer (SP) to reflect the change
  2. load values back into the registers

```
lis $3              ; 1 Increment SP
.word 8             ;   by 8
add $30,$30,$3
lw $3,-8($30)       ; 2 Load values back
lw $2,-4($30)       ;   into registers.
jr $31              ; return
```

Calling and Returning from a Subroutine

Calling a Subroutine

- to call a subroutine *jump to the memory location where the routine is located* and starting executing the code there, e.g.

```
0x00        lis $5              ; store addr of
0x04        .word func          ; label func in $5
0x08        jr $5               ; jump to func
0x0C        ...                 ; return HERE
   ...
func:
   ...
```

- Problem: how do we know where to return?

Calling and Returning from a Subroutine

Calling a Subroutine

- *need to store current location of the PC using* `jalr` which stores the address of the next statement (0x0C) in $31

```
0x00        lis $5              ; store addr of
0x04        .word func          ; label func in $5
0x08        jalr $5             ; jump to func
0x0C        ...                 ; return HERE
   ...
func:
   ...
```

- $31 now contains the address 0x0C.
- Problem: what if $31 previously had a valid return address
  - e.g. this subroutine was called by another
  - the function is a recursive function

Calling and Returning from a Subroutine

Calling a Subroutine

Solution: save the contents of $31 in the stack

Save $31 the on stack before calling subroutine *func*

1. push $31 onto stack and update stack pointer
2. jump to subroutine *func* using **`jalr`**

…

Restore $31 after returning from subroutine *func*

1. update stack pointer
2. pop value from stack and store in $31

Calling and Returning from a Subroutine

Calling a Subroutine

```
main:   sw $31,-4($30)        ; 1. push $31 onto
        lis $31               ;    the stack and
        .word 4               ;    update SP($30)
        sub $30,$30,$31       ;
        lis $5                ; 2. load addr of
        .word func            ;    subroutine func
        jalr $5               ;    and jump to it

                              ; returning from func
        lis $31               ; 1. update SP($30)
        .word 4               ;    by adding 4
        add $30,$30,$31       ;
        lw $31,-4($30)        ; 2. pop top of stack
        jr $31                ;    into $31 & return
```

Subroutines: arguments and results

Passing Arguments and Returning Results

- *Problem: need to pass arguments and return result(s)*
- can use registers, stack, or both
- need to agree between caller and callee
- *in CS 241 we will pass all parameters on the stack*
- there are other standards (e.g. CS 350)
- *the format must be documented*
- Example: create a function that will sum the first *n* natural numbers (i.e. answer = 1 + 2 + … + *n*)
- the input, *n*, is in \$2; return the answer in \$3.

Subroutines

Passing Arguments and Returning Results

1. Document your use of registers in function header

```
;; sum1toN - adds the integers 1..N
;; Registers:
;; $1 – i: which will range from 1 to N
;; $2 – N: the input
;; $3 – answer: the output
```

Subroutines

Passing Arguments and Returning Results

2. Save the current contents of any registers you are changing (except output) on the stack, in this case $1 and $2.

```
sum1toN:
   sw $1,-4($30)        ; push $1,$2 onto stack
   sw $2,-8($30)
   lis $1               ; decrement SP by 8
   .word 8
   sub $30,$30,$1
```

Subroutines

Passing Arguments and Returning Results

3. Initialize i ($1) and answer ($3), then calculate sum

```
    add $3,$0,$0        ; initialize answer = 0
    lis $1              ; initialize i = 1
    .word 1
top:
    add $3,$3,$2        ; answer = answer + i
    sub $2,$2,$1        ; i = i - 1;
    bne $2,$0,top       ; loop until i = 0
```

Subroutines

Passing Arguments and Returning Results

4. Restore the previous contents of any registers you used from the stack and then return

```
lis $1              ; update stack pointer $30
.word 8             ;
add $30,$30,$1      ;
lw $2,-8($30)       ; pop register $2
lw $1,-4($30)       ; pop register $1
jr $31              ; return
```

Low Level Errors

Common Errors

- illegal instruction
  - plus \$1, \$2, \$3 ; no such opcode as plus
- assignment to read-only register
  - add \$0, \$1, \$2 ; \$0 is read only
- division by 0
  - div \$1, \$0
- alignment violation
  - lw \$1, 3(\$0) ; must be a multiple of 4
- and possibly others...
- usually result in exception and termination

Debugging

- debugging assembly language programs is difficult
  - *terminate program (jr $31) at various places and study the values in the register*
  - eventually, use output to screen
- general techniques
  - analyze log output
  - controlled step-by-step execution

    ⇒ need some kind of virtual environment
  - verify assertions

Other Instructions

For the sake of completeness I'll mention that there are other instructions

- *immediate*
  - replace register operand with 16-bit constant
- *logical*
  - AND, OR, etc.
- *floats*
  - floating point arithmetic
- *jump*
  - long-range unconditional branch