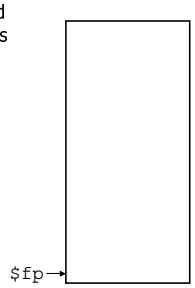
10/9: Lecture Topics

- Starting a Program
- Exercise 3.2 from H+P
- Review of Assembly Language
- RISC vs. CISC

Frame Pointer

- $sp is enough to find the saved registers and the local variables
- $fp is a **convenience**

```
foo(){
  int Array1[20];
  int i;
  ...
  for(i=0; i<20; i++){
    int Array2[20];
    ...
    for(j=0; j<20; j++){
       int Array3[20];
       ...
    }
  }
}
```



The Executable

- End result of compiling, assembling, and linking: the *executable*
- Contains:
  - Header, listing the lengths of the other segments
  - Text segment
  - Static data segment
  - Potentially other segments, depending on architecture & OS conventions

Run Time

- We'll learn a lot more about this during the OS part of the course
- In a nutshell:
  - Some *dynamic linking* may occur
    - some symbols aren't defined until run time
    - Windows' dlls (dynamic link library)
    - why do this?
  - The segments are loaded into memory
  - The OS transfers control to the program and it runs

```
       add  $a1, $a1, $a1
       add  $a1, $a1, $a1
       add  $v0, $zero, $zero
       add  $t0, $zero, $zero
outer: add  $t4, $a0, $t0
       lw   $t4, 0($t4)
       add  $t5, $zero, $zero
       add  $t1, $zero, $zero
inner: add  $t3, $a0, $t1
       lw   $t3, 0($t3)
       bne  $t3, $t4, skip
       addi $t5, $t5, 1
skip:  addi $t1, $t1, 4
       bne  $t1, $a1, inner
       slt  $t2, $t5, $v0
       bne  $t2, $zero, next
       add  $v0, $t5, $zero
       add  $v1, $t4, $zero
next:  addi $t0, $t0, 4
       bne  $t0, $a1, outer
```

```
       a1 = a1 + a1
       a1 = a1 + a1
       v0 = 0
       t0 = 0
outer: t4 = a0 + t0
       t4 = t4[0]
       t5 = 0
       t1 = 0
inner: t3 = a0 + t1
       t3 = t3[0]
       if(t3!=t4) goto skip
       t5 = t5 + 1
skip:  t1 = t1 + 4
       if(t1!=a1) goto inner
       t2 = (t5 < v0)
       if(t2!=0) goto next
       v0 = t5 + 0
       v1 = t4 + 0
next:  t0 = t0 + 4
       if(t0!=a1) goto outer
```

```
        a1 = a1 * 4

        v0 = 0
        t0 = 0
outer:      t4 = a0[t0]

            t5 = 0
            t1 = 0
inner:          t3 = a0[t1]

                if( t3 == t4 )
                    t5++
                t1 = t1 + 4
                if(t1!=a1) goto inner

            if( t5 >= v0 )
                v0 = t5
                v1 = t4
            t0 = t0 + 4
            if(t0!=a1) goto outer
```

```
a1 = a1 * 4

v0 = 0
t0 = 0
while( t0 != a1 ) {
    t4 = a0[t0]

    t5 = 0
    t1 = 0
    while( t1 != a1 ) {
        t3 = a0[t1]

        if( t3 == t4 )
            t5++
        t1 = t1 + 4
    }
    if( t5 >= v0 )
        v0 = t5
        v1 = t4

    t0 = t0 + 4
}
```

```
a1 = a1 * 4

v0 = 0
t0 = 0
while( t0 != a1 ) {
    t5 = 0
    t1 = 0
    while( t1 != a1 ) {
        if( a0[t1] == a0[t0] )
            t5++
        t1 = t1 + 4
    }
    if( t5 >= v0 )
        v0 = t5
        v1 = a0[t0]
    t0 = t0 + 4
}
```

```
maxCount = 0
i = 0
while( i != size ) {
    count = 0
    j = 0
    while( j != size ) {
        if( values[j] == values[i] )
            count++
        j++
    }
    if( count >= maxCount )
        maxCount = count
        v1 = values[i]
    i++
}
```

```
int size = 5000;
int values[size];
int mode = 0;
int modeCount = 0;
int i,j,count;

for( i = 0; i < size; i++ ) {
  count = 0;
  for( j = 0; j < size; j++ ) {
    if( values[i] == values[j] ) {
      count++;
    }
  }
  if( count >= modeCount ) {
    modeCount = count;
    mode = values[i];
  }
}
```

Assembly Language

- Architecture vs. Organization
- Machine language is the sequence of 1's and 0's that the CPU executes
- Assembly is an ascii representation of machine language
- A long long time ago, everybody programmed assembly
- Programmers that interface with the raw machine still use it
  - compiler/OS writers, hardware manufacturers
- Other programmers still need to know it
  - understand how the OS works
  - understand how to write efficient code

MIPS Instructions and Addressing

- Types of MIPS instructions
  - arithmetic operations (add, addi, sub)
    - operands must be registers or immediates
  - memory operations
    - lw/sw, lb/sb etc., only operations to access memory
    - address is a register value + an immediate
  - branch instructions
    - conditional branch (beq, bne)
    - unconditional branch (j, jal, jr)
- MIPS addressing modes,
  - register, displacement, immediate, pc-relative, pseudodirect

Accessing Memory

- Memory is a big array of bytes and the address is the index
- Addresses are 32-bits (an address is the same as a pointer)
- A word is 4 bytes
- Accesses must be aligned

Representing Numbers

- Bits have no inherent meaning
- Conversion between decimal, binary and hex
- 2's complement representation for signed numbers
  - makes adding signed and unsigned easy
  - flip the bits and add 1 to convert +/-
- floating point representation
  - sign bit, exponent (w/ bias), and significand (leading 1 assumed)

Instruction Formats

- The opcode (1st six bits of instruction) determines which format to use
- R (register format)
  - add $r0, $r1, $r2
  - [op, rs, rt, rd, shamt, func]
- I (immediate format)
  - lw $r0, 16($t0)
  - addi $r0, $t1, -14
  - [op, rs, rt, 16-bit immediate]
- J (jump format)
  - jal ProcedureFoo
  - [op, address]

MIPS Registers

- Integer registers hold 32-bit values
  - integers, addresses
- Purposes of the different registers
  - $s0-$s7
  - $t0-$t9
  - $a0-$a3
  - $v0-$v1
  - $sp, $fp
  - $ra
  - $zero
  - $at
  - $k0-$k1
  - $gp

Procedure Calls

- Calling the procedure is a jump to the label of the procedure
  - "jal ProcedureName" what happens?
- Returning from a procedure is a jump to the instruction after "jal ProcedureName"
  - "jr $ra"
- Arguments are passed in $a0-$a3
- Return values are passed back in $v0-$v1
- $t0-$t9 and the stack is used for local variables
- Registers are saved on the stack, but not automatically
  - if necessary, caller saves $t0-$t9, $a0-$a3, $ra
  - if necessary, callee saves $s0-$s7

Understanding Assembly

- Understand the subset of assembly that we've used, know what the instructions do
- Trace through assembly language snippets
- Assembly → C, recognize
  - while loops, if statements, procedure calls
- C → Assembly, convert
  - while loops, if statements, procedure calls

RISC vs. CISC

- *Reduced Instruction Set Computer*
  - MIPS: about 100 instructions
  - Basic idea: compose simple instructions to get complex results
- *Complex Instruction Set Computer*
  - VAX: about 325 instructions
  - Basic idea: give programmers powerful instructions; fewer instructions to complete the work

The VAX

- Digital Equipment Corp, 1977
- Advances in microcode technology made complex instructions possible
- Memory was expensive
  - Small program = good
- Compilers had a long way to go
  - Ease of translation from high-level language to assembly = good

VAX Instructions

- Queue manipulation instructions:
  - INSQUE: insert into queue
- Stack manipulation instructions:
  - POPR, PUSHR: pop, push registers
- Procedure call instructions
- Binary-encoded decimal instructions
  - ADDP, SUBP, MULP, DIVP
  - CVTPL, CVTLP (conversion)

The RISC Backlash

- Complex instructions:
  - Take longer to execute
  - Take more hardware to implement
- Idea: compose simple, fast instructions
  - Less hardware is required
  - Execution speed may actually *increase*
- PUSHR vs. sw + sw + sw

How many instructions?

- How many instructions do you *really* need?
- Potentially only one: subtract and branch if negative (`sbn`)
- See p. 206 of your book