Numbers: positional notation

- Number Base $B \Rightarrow B$ symbols per digit:
  - Base 10 (Decimal): 0 ~ 9
  - Base 2 (Binary): 0, 1
  - Base 16 (Hexadecimal): 0 ~ 9, A ~ F
  - Base 8 (Octal): 0 ~ 7

- Number representation:
  - $d_31d_{30} \ldots d_2d_1d_0$ is a 32 digit number
  - value = $d_31 \times B^{31} + \ldots + d_1 \times B^1 + d_0 \times B^0$

- Conversion: Binary <-> Hex, Oct
  - 1 hex digit replaces 4 binary digits
  - 1 oct digit replaces 3 binary digits

Example:
- 1010 1100 0101 (binary) = ? (hex), ? (oct)

How to Represent Negative Numbers?

- Sign and magnitude
  - One bit for sign, rest of bits for numerical value
  - Arithmetic circuit more complicated
  - Two zeros

- One's complement
  - The negative number representation is a negation
  - Arithmetic circuit more complicated
  - Two zeros

- Two's complement
  - $d_31 \times 2^{31} + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$
  - Same arithmetic circuit for non-negative and negative
  - One zeros

- Used by most computers

Two’s complement shortcut

- Negation
  - Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result

- Sign extension
  - Convert 2’s complement number using $n$ bits to more than $n$ bits
  - Simply replicate the most significant bit (sign bit) of smaller to fill new bits

- Overflow
  - The result of an arithmetic operation cannot be represented by these rightmost HW bits
  - What is the condition?

Pointers

- Pointer: A variable that contains the address of a variable.

- Pointer operations
  - & operator: get address of a variable
  - * "dereference operator": get value pointed to

- Normally a pointer can only point to one type (int, char, a struct, etc.).

- void * is a type that can point to anything (generic pointer)

Pointers and Parameter Passing

- Java and C pass a parameter “by value”
  - procedure/function gets a copy of the parameter, so changing the copy cannot change the original

- How to get a function to change a value?
  - Use pointer variable

```c
void addOne(int x) { void addOne(int *x) {
  x = x + 1; *x = *x + 1;
} => }
int y = 3; int y = 3;
addOne(y); addOne(&y);
```
**Pointer Arithmetic**

- Since a pointer is just a memory address, we can add to it to traverse an array.
- \( \text{ptr}+1 \) will return a pointer to the next array element.
- \( *\text{ptr}+1 \) vs. \( *\text{ptr}++ \) vs. \( *(\text{ptr}+1) \) ?
- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, \( \text{ptr}+1 \) doesn’t add 1 to the memory address, but rather adds the size of the array element.

**Pointers & Allocation**

- After declaring a pointer:
  
  ```c
  int *ptr;
  ```

  \( \text{ptr} \) doesn’t actually point to anything yet. We can either:
  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to... (next time)

**Arrays (1/2)**

- Arrays are (almost) identical to pointers
  - char *string and char string[] are nearly identical declarations
  - They differ in very subtle ways: incrementing, declaration of filled arrays
- **Key Concept:** An array variable is a pointer to the first element.
- **Consequences:**
  - \( \text{ar} \) is a pointer
  - \( \text{ar}[0] \) is the same as \( *\text{ar} \)
  - \( \text{ar}[2] \) is the same as \( *(\text{ar}+2) \)

**Arrays (2/2)**

- Declared arrays are only allocated while the scope is valid
  ```c
  char *foo() {
  char string[32]; ...;
  return string;}
  ```

  is incorrect

  - An array in C does not know its own length, & bounds not checked!
    - Consequence: We can accidentally access off the end of an array.
    - Consequence: We must pass the array and its size to a procedure which is going to traverse it.

**C Strings**

- A string in C is just an array of characters.
  ```c
  char string[] = "abc";
  ```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (null terminator)
  ```c
  int strlen(char s[]) {
  int n = 0;
  while (s[n] != 0) n++;
  return n;
  }
  ```

- One common mistake is to forget to allocate an extra byte for the null terminator.

**Pointers to pointers**

- But what if what you want changed is a pointer?
- **Solution!** Pass a pointer to a pointer, called a handle, declared as \( **h \)
C structures: Overview

- A struct is a data structure composed for simpler data types.
  ```c
  struct point {
    int x;
    int y;
  }
  ```

- The C arrow operator (->) dereferences and extracts a structure field with a single operator.

- The following are equivalent:
  ```c
  struct point *p;
  printf("x is %d\n", (*p).x);
  printf("x is %d\n", p->x);
  ```

Sizeof() operator

- C has operator sizeof() which gives size in bytes (of type or variable)

- Assume size of objects can be misleading, so use sizeof(type)

- How big is sizeof(p)?
  ```c
  struct p {
    char x;
    int y;
  };
  ```

  - 5 bytes? 8 bytes?
  - Compiler may word align integer y

Dynamic Memory Allocation

- To allocate room for something new to point to, use malloc() (with the help of a typecast and sizeof):
  ```c
  ptr = (int *) malloc (sizeof(int));
  ```

- Once malloc() is called, use the memory location only after setting its value (due to garbage).

- After dynamically allocating space, we must dynamically free it:
  ```c
  free(ptr);
  ```

Where allocated?

- Structure declaration does not allocate memory
- Variable declaration does allocate memory
  - If declare outside a procedure, allocated in static storage
  - If declare inside procedure, allocated on the stack and freed when procedure returns

C Memory Management

- C has 3 pools of memory
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address (location of "activation records" in Java or "stack frame" in C)
  - The Heap (dynamic storage): data lives until deallocated by programmer

- C requires knowing where objects are in memory, otherwise don't work as expected
  - Java hides location of objects

Memory Management

- How do we manage memory?
- Code, Static storage are easy: they never grow or shrink
- Stack space is also easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time
Choosing a block in malloc()

° If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
  • best-fit: choose the smallest block that is big enough for the request
  • first-fit: choose the first block we see that is big enough
  • next-fit: like first-fit but remember where we finished searching and resume searching from there

Internal vs. External Fragmentation

° With the slab allocator, difference between requested size and next power of 2 is wasted
  • e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.
° We also refer to this as fragmentation, but call it internal fragmentation since the wasted space is actually within an allocated block.
° External fragmentation: wasted space between allocated blocks.

Assembly Variables: Registers

° Assembly Operands are registers
  • limited number of special locations built directly into the hardware
  • operations can only be performed on these!
° Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)
° Drawback: Since registers are in hardware, there are a predetermined number of them
° 32 registers in MIPS

Each MIPS register is 32 bits wide

Register Zero

° One particular immediate, the number zero (0), appears very often in code.
° So we define register zero ($0 or $zero) to always have the value 0; eg
  add $s0,$s1,$zero (in MIPS)
  \[ f = g \] (in C)
  where MIPS registers \( s0, s1 \) are associated with C variables \( f, g \)
° defined in hardware, so an instruction
  addi $zero,$zero,5

will not do anything!

Immediates

° Immediates are numerical constants.
° They appear often in code, so there are special instructions for them.
° Add Immediate:
  addi $s0,$s1,10 (in MIPS)
  \[ f = g + 10 \] (in C)
  where MIPS registers \( s0, s1 \) are associated with C variables \( f, g \)
° Syntax similar to add instruction, except that last argument is a number instead of a register.

Assembly Operands: Memory

° C variables map onto registers; what about large data structures like arrays?
° 1 of 5 components of a computer: memory contains such data structures
° But MIPS arithmetic instructions only operate on registers, never directly on memory.
° Data transfer instructions transfer data between registers and memory:
  • Memory to register
  • Register to memory
Data Transfer: Memory to Reg (1/2)

To specify a memory address to copy from, specify two things:
- A register which contains a pointer to memory
- A numerical offset (in bytes)

The desired memory address is the sum of these two values.

Example: 8($t0)
- specifies the memory address pointed to by the value in $t0, plus 8 bytes

Data Transfer: Memory to Reg (2/2)

Example:
```
1w $t0, 12($s0)
```
This instruction will take the pointer in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0

Notes:
- $s0 is called the base register
- 12 is called the offset
- offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure

Data Transfer: Reg to Memory

Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- MIPS Instruction Name: `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)

Example:
```
sw $t0, 12($s0)
```
This instruction will take the pointer in $s0, add 12 bytes to it, and then store the value from register $t0 into the memory address pointed to by the calculated sum

Pointers v. Values

Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), and so on

If you write `add $t2, $t1, $t0`
then $t0 and $t1 better contain values

If you write `lw $t2, 0($t0)`
then $t0 better contain a pointer

Don’t mix these up!

Addressing: Byte vs. word

Every word in memory has an address, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - Memory[0], Memory[1], Memory[2], ...
    Called the address of a word
- Computers needed to access 8-bit bytes as well as words (4 bytes/word)
- Today machines address memory as bytes, hence 32-bit (4 byte) word addresses differ by 4

Compilation with Memory

What offset in `lw` to select A[8] in C?
- 4x8=32 to select A[8]: byte v. word
- Compile by hand using registers:
  g = h + A[8];
  - g; $s1, h; $s2, $s3: base address of A
- 1st transfer from memory to register:
  lw $t0, 32($s3) # $t0 gets A[8]
  - Add 32 to $s3 to select A[8], put into $t0
- Next add it to h and place in g:
  add $s1, $s2, $t0 # $s1 = h+A[8]
Role of Registers vs. Memory
° What if more variables than registers?
  • Compiler tries to keep most frequently used variable in registers
  • Less common in memory: spilling
° Why not keep all variables in memory?
  • Smaller is faster: registers are faster than memory
  • Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

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;
  goto L2;
  L1:
  clause1;
  L2:
  Not as elegant as if-else, but same meaning

MIPS branch instructions
° Conditional branches:
  • beq register1, register2, L1
    - if (register1==register2) goto L1
  • bne register1, register2, L1
    - if (register1!=register2) goto L1
° Unconditional branch:
  j label
  • jump (or branch) directly to the given label
  • Same meaning as (using C):
    goto label

Compiling C if into MIPS
° Compile by hand
  if (i == j) f=g+h;
  else f=g-h;
° Final compiled MIPS code:
  beq $s3,$s4,True # branch i==j
  sub $s0,$s1,$s2 # f=g-h(false)
  j Fin # goto Fin
  True: add $s0,$s1,$s2 # f=g+h (true)
  Fin:

Loops in C/Assembly (1/2)
° Simple loop in C; A[] is an array of ints
  do {
    g = g + A[i];
    i = i + j;
  } while (i != h);
° Rewrite this as:
  Loop: g = g + A[i];
  i = i + j;
  if (i != h) goto Loop;
° Use this mapping:
  g: $s1, h: $s2, i: $s3, j: $s4, base of A:$s5

Loops in C/Assembly (2/2)
° Final compiled MIPS code:
  Loop: sll $t1,$s3,2
  add $t1,$t1,$s5 #f = f + A[i]
  lw $t1,0($t1) #t1 = addr A
  add $s1,$s1,$t1 #g = g + A[i]
  add $s3,$s3,$s4 #i = i + j
  bne $s3,$s2,Loop # if i!=h
**Inequalities in MIPS**

- Create a MIPS Inequality Instruction:
  - “Set on Less Than”: `slt reg1, reg2, reg3`
  - Meaning:
    ```
    if (reg2 < reg3)
        reg1 = 1;
    else reg1 = 0;
    ```
- How do we implement >, <= and >= ?
  - Use BNE, BEQ or change reg2 and reg3
- There is also an immediate version of `slt` to test against constants: `slti`

**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.

**Rules for Procedures**

- 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?

**MIPS Registers**

- The constant 0 $0 $zero
- Reserved for Assembler $1 $at
- Return Values $2-$3 $v0-$v1
- Arguments $4-$7 $a0-$a3
- Temporary $8-$15 $t0-$t7
- Saved $16-$23 $s0-$s7
- More Temporary $24-$25 $t8-$t9
- Used by Kernel $26-27 $k0-$k1
- Global Pointer $28 $gp
- Stack Pointer $29 $sp
- Frame Pointer $30 $fp
- Return Address $31 $ra

(From COD 2nd Ed. p. A-23)

Use names for registers -- code is clearer!

**Register Conventions**

- Note that, if the callee is going to use some s registers, it must:
  - save those s registers on the stack
  - use the registers
  - restore s registers from the stack
  - jr $ra
- With the temp registers, the callee doesn’t need to save onto the stack.
- The caller must save those temp registers that it would like to preserve though the call.

**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
  ```
  lw $ra, framesize-4($sp)# restore $ra
  addi $sp,$sp, framesize
  jr $ra
  ```
Example: Fibonacci Numbers 1/2

The Fibonacci numbers are defined as follows: \(F(n) = F(n - 1) + F(n - 2),\) \(F(0)\) and \(F(1)\) are defined to be 1

Rewriting this in C we have:

```c
int fib(int n) {
    if(n == 0) { return 1; }
    if(n == 1) { return 1; }
    return (fib(n - 1) + fib(n - 2));
}
```

Example: Fibonacci Numbers 2/2

Here's the complete code for reference:

```c
fib:
  addi $sp, $sp, -12
  sw $ra, 8($sp)
  sw $s0, 4($sp)
  addi $v0, $zero, 1
  beq $a0, $zero, fin
  addi $t0, $zero, 1
  beq $a0, $t0, fin
  addi $a0, $a0, -1
  sw $a0, 0($sp)
  jal fib
  lw $a0, 0($sp)
  addi $a0, $a0, -1
  add $s0, $v0, $zero
  jal fib
  add $v0, $v0, $s0
  fin:
  lw $s0, 4($sp)
  lw $ra, 8($sp)
  addi $sp, $sp, 12
  jr $ra
```

Bitwise Operations

Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.

Introduction two new classes of instructions:

- Logical Operators
- Shift Instructions

Logical Operators

- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit (masking).
- Similarly, note that oring a bit with 1 produces a 1 at the output while oring a bit with 0 produces the original bit.

Shift Instructions (1/3)

- Move (shift) all the bits in a word to the left or right by a number of bits.

- MIPS shift instructions:
  1. **sll** (shift left logical): shifts left and fills emptied bits with 0s
  2. **srl** (shift right logical): shifts right and fills emptied bits with 0s
  3. **sra** (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (2/3)

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - \(11_2 \times 10_2 = 110_2\)
    - \(1010_2 \times 10_2 = 10100_2\)
  - Multiplying by 4 is same as shifting left by 2:
    - \(11_2 \times 100_2 = 1100_2\)
    - \(1010_2 \times 100_2 = 101000_2\)
  - Multiplying by \(2^n\) is same as shifting left by \(n\)
Shift Instructions (3/3)

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:
  
  ```
  a *= 8; (in C)
  would compile to:
  sll $s0,$s0,3 (in MIPS)
  ```

- Likewise, shift right to divide by powers of 2
  - remember to use `sra`

Loading, Storing bytes

- In addition to word data transfers (lw, sw), MIPS has byte data transfers:
  - `load byte: lb, lbu`
    - `lb`: sign extends to fill upper 24 bits
    - `lbu`: zero fills upper 24 bits
  - `store byte: sb`
    - same format as lw, sw

Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers.
  2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs
- What if we modify the numbers for instructions?
  - It modifies the program code (self-modifying code)

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Instructions as Numbers

- One word is 32 bits, so divide instruction word into "fields".
- Each field tells computer something about instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format (next lecture)

Instruction Formats

- I-format: used for instructions with immediates, lw and sw (since the offset counts as an immediate), and the branches (beq and bne),
  - (but not the shift instructions; later)
- J-format: used for j and jal (see Friday)
- R-format: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way.