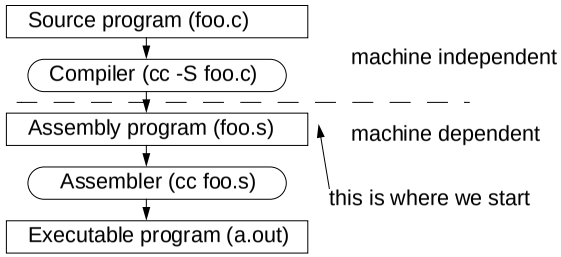
Introduction to the MIPS ISA

- Reading: H&P Ch 3 (For now, skip the material on procedures), H&P Ch 4.1-4.3, Appendix A

Overview

- Remember that the machine only understands very basic instructions (machine instructions)
- It is the compiler's job to translate your high-level (e.g. C program) into machine instructions. In more detail (forgetting linking):



- Assembly language is a thin veneer over machine language.

MIPS is a RISC

- RISC = Reduced (Regular/Restricted) Instruction Set Computer
- The MIPS ISA provides only a few (very) basic kinds of instructions, and they all have a very regular format. E.g.
  - All arithmetic operations are of the form:

R_d <- R_s op R_t `# the Rs are registers`

- Other restrictions:
  - All MIPS instructions are 32 bits long
  - The ALU can only operate on registers (*load-store architecture*).
- Why do you think the designers made these restrictions?

MIPS is a Load-Store Architecture

- Every operand of a MIPS instruction must be in a register (with some exceptions)
- Variables must be loaded into registers
- Results have to be stored back into memory
- Example C fragment...

```
a = b + c;
```

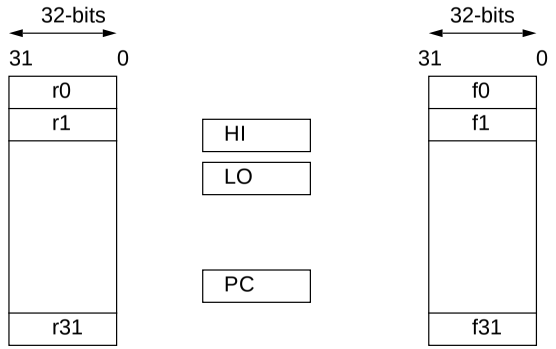
- ... would be "translated" (compiled) into something like:

```
Load b into register Rx
Load c into register Ry
Rz <- Rx + Ry
Store Rz into a
```

MIPS Registers

- Provides thirty-two, 32-bit registers, named $0, $1, $2 .. $31 used for:
  - integer arithmetic
  - address calculations
  - special-purpose functions defined by convention
  - temporaries
- A 32-bit program counter (PC)
- Two 32-bit registers HI and LO used specifically for multiplication and division (see below)
- Thirty-two 32-bit registers $f0, $f1, $f2 .. $f31 used for floating point arithmetic (we won't worry about these).

Registers in Pictures



MIPS Register Names and Conventions

Register	Name	Function	Comment
$0	zero	Always 0	No-op on write
$1	$at	reserved for assembler	don't use it!
$2-3	$v0-v1	expression eval./function return	
$4-7	$a0-a3	proc/funct call parameters	
$8-15	$t0-t7	volatile temporaries	not saved on call
$16-23	$s0-s7	temporaries (saved across calls)	saved on call
$24-25	$t8-t9	volatile temporaries	not saved on call
$26-27	$k0-k1	reserved kernel/OS	don't use them
$28	$gp	pointer to global data area	
$29	$sp	stack pointer	
$30	$fp	frame pointer	
$31	$ra	proc/funct return address	

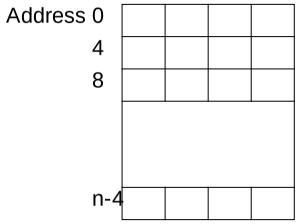
MIPS Information Units

- Data types and size:
  - Byte
  - Half-word (2 bytes)
  - Word (4 bytes)
  - Float (4 bytes, single precision format)
  - Double (8 bytes, double precision format)
- Memory is byte addressable.
- A data type must start on an address divisible by its size (in bytes)
- The address of the data type is the address of its lowest byte (MIPS on DEC is little endian)

MIPS Addressing

- In MIPS (and most byte addressable machines) every word (should) start at an address divisable by 4:

A memory of size N bytes



- This is called “aligned memory” - it makes for more efficient transfers between memory and the CPU

MIPS Instruction Types

- There are very few basic kinds of operations:
  1. Arithmetic (addition, substraction, etc)
  2. Control (branches, jumps, etc)
  3. Memory access (load and store)
  4. Comparison (less-than, greater-than, etc)
  5. Logical (and, or, xor, etc)
- We’ll use the following notation when describing instructions:

rd: destination register (modified by instruction)
rs: source register (read by instruction)
rt: source/destination register (read or read+modified)
immed: a 16-bit value

Running Example

- Let’s translate this simple C program into MIPS assembly code:

```
int x, y;

void main ()
{
  ...
  x = x + y;
  if (x==y)
    x = x + 3;
  x = x + y + 17;
  ...
}
```


Assembly Program Structure

- An assembly program consists of “data” (global data) and “text” (instructions).
- Keywords starting with a “.” are assembler directives.

```
    .data      # start of data segment
x:  .word      # data layout directive
y:  .word      # allocate two words, name them
               # x and y

    .text       # start of text segment
    .globl main # main is a “global” name
main:
# instructions go here....
```

Loading and Storing Data

- Data is explicitly moved between memory and registers through load and store instructions.
- Each load or store must specify the memory address of the memory data to be read or written.
- Think of a MIPS address as a 32-bit, unsigned integer.
- Because a MIPS instruction is always 32 bits long, the address must be specified in a more compact way.
- We always use a *base register* to address memory
- The base register points somewhere in memory, and the instruction specifies the register number, and a 16-bit, *signed offset*
- A single base register can be used to access any byte within ??? bytes from where it points in memory.

Load and Store Instructions

- Load a word from memory:

```
lw  rt, offset(base)  # rt <- memory[base+offset]
```

- Store a word into memory:

```
sw rt, offset(base)  # memory[base+offset] <- rt
```

- How do we get an address into a base register?
- Fortunately, the assembler takes care of this by providing a special “pseudo-instruction” to load addresses:

```
la  rd, Label  # rd <- address of Label
```


Arithmetic Instructions

```
Opcode  Operands       Comments

ADD     rd, rs, rt     # rd <- rs + rt
ADDI    rt, rs, immed  # rt <- rs + immed
SUB     rd, rs, rt     # rd <- rs - rt

Examples:

ADD     $8, $8, $10    # r8 <- r9 + r10
ADD     $t0, $t1, $t2  # t0 <- t1 + t2
SUB     $s0, $s0, $s1  # s0 <- s0 - s1
ADDI    $t3, $t4, 5    # t3 <- t4 + 5
```


Example

```
x = x + y;

if (x==y)

   x = x + 3;

x = x + y + 17;
```