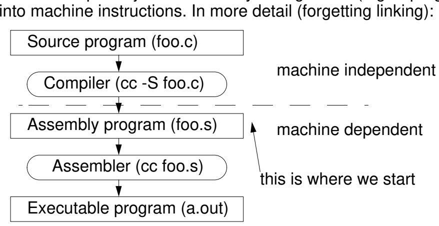
Introduction to the MIPS ISA

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

Overview (2)

- Think about a simple C program...

```
int array[100];
void main () {
    int i;
    for (i=0; i<100; i++)
        array[i] = i;
}
```

- What set of instructions (ISA) should the machine provide to execute it?
- What does your intuition tell you about trade-offs between the ISA and the size (length) of the resulting machine program?
- What kind of trade-offs exist between the ISA and the speed, cost, complexity of the hardware needed to execute the program?
- Answering these questions are some of the challenges of ISA design.

Review of Binary/Hex Representation

- Remember that computers represent all data (integers, floating point numbers, characters, instructions, etc.) in a binary representation.
- Representing integers: What characteristics does our scheme need?
  - Easy test for positive/negative.
  - Equal number of positive and negative numbers
  - Easy check for overflow
- Different schemes: sign and magnitude, 1's complement, 2's complement
- 2's complement tricks (sign bit extension, converting from positive to negative, addition/subtraction)
- Hexidecimal notation
- Common powers of 2 (10:1024 (1K), 20:(1M), 30(1G),8:256, 16: 64K, 32(4G))

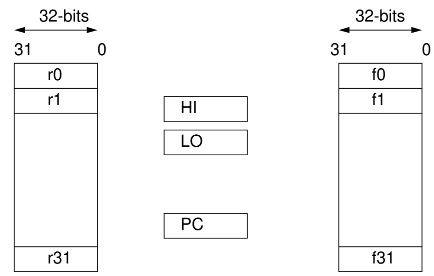
MIPS ISA Overview

- MIPS is a "computer family": R2000/3000 (32-bit), R4000/4400 (64-bit)
- New entries include R8000 (scientific/graphics) and R10000
- MIPS originated as a Stanford project: **M**icroprocessor without **I**nterlocked **P**ipe **S**tages
- All MIPS instructions are 32 bits long.
- H+P posit 4 principles of hardware design. Try to keep them in mind during our discussion of the MIPS ISA:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Compromise
  4. Make the common case fast

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
- Thirty-two 32-bit registers $f0, $f1, $f2 .. $f31 used for floating point arithmetic
- Other special-purpose registers (see later)

Registers are part of the process “state”



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 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;
d = a + b;
```

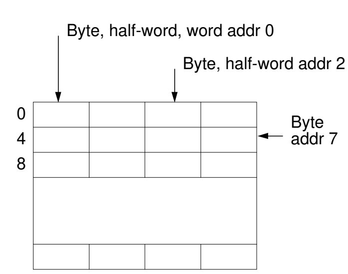
- ... would be “translated” into something like:

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


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)

Addressing Revisited



MIPS Instruction Types

- As we said earlier, there are very few basic operations :
  1. Arithmetic (addition, substraction, etc)
  2. Logical (and, or, xor, etc)
  3. Comparison (less-than, greater-than, etc)
  4. Control (branches, jumps, etc)
  5. Memory access (load and store)
- 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

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
```


Integer Arithmetic

- Numbers can be either *signed* or *unsigned*
- If the numbers are signed, they are represented in 2's complement notation (Ch. 4, Sections 1-3)
- There are instructions for (+, -, /, *) for signed or unsigned integers, for example:
  - ADD and ADDU
  - ADDI and ADDIU
  - MULT and MULTU
- For add and subtract, the computation is the same for both, but the machine will signal an overflow when one occurs for signed numbers.

Overflows in 2's Complement

- Overflow occurs when the addition of two numbers of the same sign results in a sum of the opposite sign
- Overflow cannot occur when adding operands of different signs
- Example 1: Assume a 4-bit machine. Register 9 contains 7 and register 10 contains 3
  - r9 = 0111 r10 = 0011
  - ADD $11, $9, $10 yields r11 = 1010, which has the opposite sign, so the machine will signal overflow
  - ADDU $11, $9, $10 will yield the same result, but no overflow is signaled
- Example 2: Assume a 4-bit machine. Register 9 contains 7 and register 10 contains -3
  - r9 = 0111 r10 = 1101
  - ADD $11, $9, $10 yields r11 = 0100, which is hunky dory
  - ditto for ADDU

How does the machine know if the number is signed or unsigned?

- It doesn't.
- You (or the compiler) should!
- You have to tell the machine by using the right instruction.

Multiply and Divide Instructions

- Multiplying two 32-bit numbers can yield a 64 bit number. Hence the use of HI and LO registers.
- Dividing two numbers yields a quotient and a remainder.

```
Opcode  Operands        Comments

MULT    rs, rt          # HI/LO <- rs * rt
MULTU   rs, rt          # HI/LO <- rs * rt
DIV     rs, rt          # LO <- rs/rt
                        # HI <- rs rem rt
DIVU    rs, rt          # LO <- rs/rt
                        # HI <- rs rem rt
```

- If an operand is negative, the remainder is not specified by the MIPS architecture.

Multiply and Divide Instructions (2)

- There are instructions to move between HI/LO registers.

```
Opcode  Operands       Comments

MFHI    rd             # rd <- HI
MTHI    rs             # HI <- rs
```


Load and Store Instructions

- 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 Examples

- 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
```

- For smaller units (bytes, half-words) only the lower bits of a register are accessable. Also, for loads, you need to specify whether to sign or zero extend the data.

```
lb  rt, offset(base) # rt <- sign-extended byte
lbu rt, offset(base) # rt <- zero-extended byte
sb  rt, offset(base) # store low order byte of rt
```


Flow of Control: Conditional Branches

- You can compare on...
  - equality or inequality of two registers
  - comparison of register to zero (>, <, <=, >=)
- ... and branch to a target that is a signed displacement (expressed in number of instructions [words not bytes!]) from the instruction *following* the branch.
- In assembly language, it's easiest to just use the target address (from the label), rather than trying to figure out the number of instructions.

```
BEQ     rs, rt, target # branch if rs == rt
BNE     rs, rt, target # branch if rs != rt
BGTZ    rs, target     # branch if rs > 0
BGEZ    rs, target     # branch if rs >= 0
BLTZ    rs, target     # branch if rs < 0
BLEZ    rs, target     # branch if rs <= 0
```


Comparison Between Registers

- What if you want to branch if R6 is greater than R7?
- We can use the SLT instruction:

```
SLT     rd, rs, rt     # if rs<rt then rd <- 1
                       #   else rd <- 0
SLTU    rd, rs, rt     # same, but rs,rt unsigned
```

- Example: Branch to L1 if $5 > $6

```
SLT     $7, $6, $5   # $7 = 1, if $6 < $5
BNE     $7, $0, L1
```


Jump Instructions

- Jump instructions allow for unconditional transfer of control:

```
J       target      # go to specified target
JR      rs          # jump to addr stored in rs
```

- Jump and link is used for procedure calls:

```
JAL     target      # jump to target, $31 <- PC
JALR    rs, rd      # jump to addr in rs
                    # rd <- PC
```

- When calling a procedure, use JAL; to return, use JR $31

Logic Instructions

- Used to manipulate bits within words, set up masks, etc.

```
Opcode  Operands        Comments

AND     rd, rs, rt      # rd <- AND(rs, rt)
ANDI    rt, rs, immed   # rt <- AND(rs, immed)
OR      rd, rs, rt
ORI     rt, rs, immed
XOR     rd, rs, rt
XORI    rt, rs, immed
```

- The immediate constant is limited to 16 bits
- To load a constant in the 16 upper bits of a register we use LUI:

```
Opcode  Operands        Comments

LUI     rt, immed       # rt<31,16> <- immed
                        # rt<15,0>  <- 0
```


Logic Instruction Examples

1. Turn *on* the bits in the low order byte of R6:

```
ORI     $6, $6, 0x00ff     # set r6<7,0> to 1s
```

2. Turn *off* the bits in the low order byte of R6:

```
LUI     $5, 0xffff      # set r5<31,16> to 1s
ORI     $5, 0xff00      # zero low order byte
AND     $6, $6, $5      # zap low order byte in R6
```

3. Flip the the bits in the high order byte of R6: (check this one)

```
LUI     $5, 0xff00      # 1s in upper byte
ANDI    $5, 0x0000      # 0s every where else
XOR     $6, $6, $5      # flip upper bits...
```


Shift Instructions

- Used to move bits around within registers.
- Logical shifts (zeros are shifted in from end).

```
SLL     rd, rt, shamt  # rd = rt shifted left by
                       # shamt
SRL     rd, rt, shamt  # right shift
```

- Arithmetic shift right (sign extend from left)

```
SRA     rd, rt, shamt  # rd = rt shifted right by
                       # shamt, and sign extended
```

- shamt is a 5-bit shift amount

Example C Program

```
#include <stdio.h>
int array[100];

void main ()
{
  int i;
  for (i=0; i<100; i++)
    array[i] = i;
}
```


Same Program in Assembly

```
        .data           # begin data segment
array:  .space 400      # allocate 400 bytes

        .text           # begin code segment
        .globl main     # entry point must be global

main:   move $t0, $0    # $t0 is used as counter
        la   $t1, array # $t1 is pointer into array
start:  bge  $t0, 100, exit# more than 99 iterations?
        sw   $t0, 0($t1)  # store zero into array
        addi $t0, $t0, 1  # increment counter
        addi $t1, $t1, 4  # increment pointer into array
        j    start        # goto top of loop
exit:   j    $ra          # return to caller of main...
        .end main
```