Brief Review of the MIPS Instruction Set Architecture

RISC Instruction Set Basics

- All operations on data apply to data in registers and typically change the entire register
- The only operations that affect memory are load and store operations
- The instruction formats are few in number with all instructions typically one size

- Text uses MIPS64
  - Instructions generally have a **D** at the start or end of the mnemonic, e.g. **DADD** is 64 bit ADD

MIPS ISA

- 32 registers
  - Register 0 always has the value 0
- Three classes of instructions
  - ALU instructions
    - Register to register or immediate to register
    - Signed or unsigned
    - Floating point or Integer
    - NOT to memory
  - Load/Store instructions
    - Base register added to signed offset to get an effective address
  - Branches and Jumps
    - Branch based on condition bit or comparison between pair of registers

MIPS arithmetic

- Most instructions have 3 operands
- Operand order is fixed (destination first)

Example:

HLL code: `A = B + C;`

MIPS code: `DADD $s0, $s1, $s2`

($s0, $s1 and $s2 are associated with variables by compiler)

MIPS arithmetic

HLL code:
```
A = B + C + D;
E = F - A;
```

MIPS code:
```
DADD $t0, $s1, $s2
DADD $s0, $t0, $s3
DSUB $s4, $s5, $s0
```

Operands must be registers

- Compiler tries to keep as many variables in registers as possible
- Some variables can not be allocated
  - large arrays
  - aliased variables (variables accessible through pointers)
  - dynamically allocated variables on the heap or stack
- Compiler may run out of registers; this is called **spilling**

Memory layout: Alignment

- Words are aligned (32 bit in this example)
- Big-endian or Little-endian depending on the OS

Instructions: load and store

Example:

HLL code: `A[3] = h + A[3];`

MIPS code:

```
LW   $t0, 24($s3)
DADD $t0, $s2, $t0
SW   $t0, 24($s3)
```

- 8 bytes per word → offset to 3rd word → 24 byte displacement
- h already in register $s2
- Store word operation has no destination (reg) operand

Swap example

C

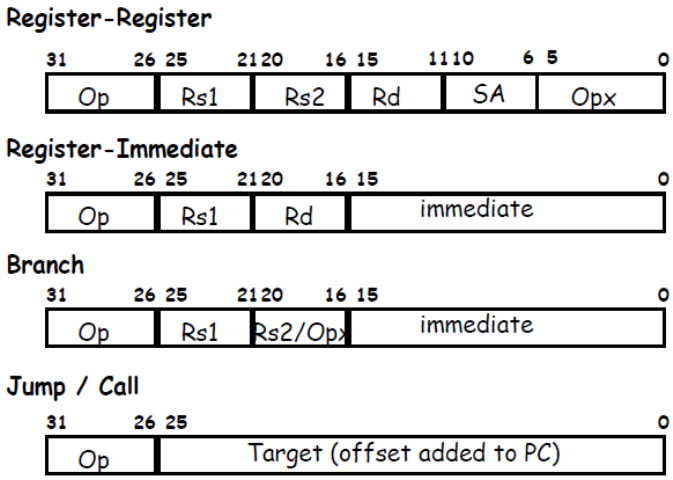
```
swap(int v[], int k)
{
   int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

MIPS32

```
swap:
   MULI $2,  $5, 4
   ADD  $2,  $4, $2
   LW   $15, 0($2)
   LW   $16, 4($2)
   SW   $16, 0($2)
   SW   $15, 4($2)
   JR   $31
```

Explanation:
index k : $5
base address of v: $4
address of v[k] is $4 + 4*$5

MIPS32 Instruction Formats



Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS unconditional branch

```
J Label
```

- MIPS conditional branch instructions:

```
BNE $t0, $t1, Label
BEQ $t0, $t1, Label
```

- Example: if (X==Y)
  A = B + C;

```
        BNE  $s4, $s5, Label
        DADD $s3, $s0, $s1
Label:  ....
```

Assembler calculates offset amount for us

Control Flow

- We have: BEQ, BNE, what about Branch-if-less-than?
- New instruction:

```
SLT $t0, $s1, $s2
```

meaning:

```
if  $s1 < $s2 then
   $t0 = 1
else
   $t0 = 0
```

- Can follow this with **`BNE $t0, $zero, Label`**
  to get branch if less than

MIPS compiler conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved (by callee)
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

What's this do? 32 bits

```
    LI     $3,  4              # load immediate
Foo:
    MULI   $2,  $3, 4
    ADD    $2,  $1, $2
    LW     $15, 0($2)
    ADDI   $15, $15, 1
    SW     $15, 0($2)
    ADDI   $3,  $3, -1
    BNE    $3, $zero, Foo
```

Brief look at the 80x86

- Textbook appendix has more details
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow
- Implementation on later processors translates x86 instructions into RISC-like instructions internally, allowing it to adopt many of the RISC innovations