6.S078 - Computer Architecture:
A Constructive Approach

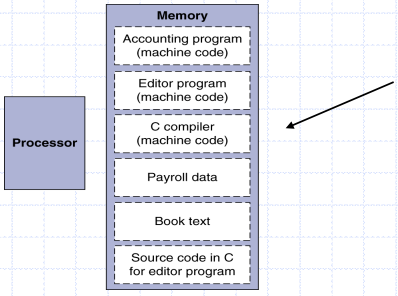
Introduction to SMIPS

Li-Shiuan Peh
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Stored Program Concept

- Instructions are bits (i.e., as numbers)
- Programs are stored in memory
  - to be read or written just like data

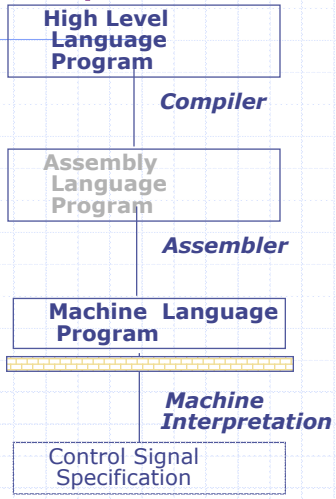
Treating Instructions in the same way as Data



memory for data, programs, compilers, editors, etc.

- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the next instruction and continue

Multiple Levels of Representation




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw      $15, 0($2)
lw      $16, 4($2)
sw$16, 0($2)
sw$15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK

High and low signals on control lines

Instruction Set Architecture (ISA)

- Programmer's view of the computer
  - Instructions, operands

Example ISAs

- Intel 80x86
- ARM
- IBM/Motorola PowerPC
- HP PA-RISC
- Oracle/Sun Sparc
- 6.004's Beta

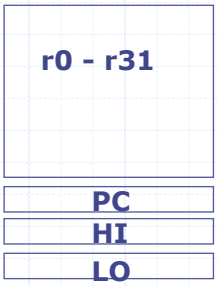
Why MIPS?

It's simple!
Most taught ISA

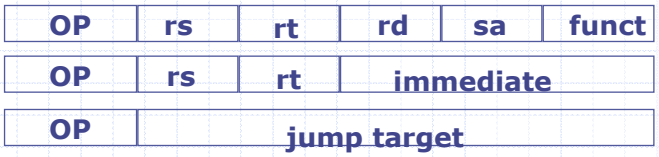
MIPS I Instruction Set Architecture

- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers



3 Instruction Formats: all 32 bits wide



- **SMIPS: a subset of the full MIPS32 ISA**

SMIPS Registers: Fast Locations for Data

- 32 32-bit registers: $0, $1, … , $31
  - operands for integer arithmetic
  - address calculations
  - temporary locations
  - special-purpose functions defined by convention
- 1 32-bit Program Counter (PC)
- 2 32-bit registers HI & LO:
  - used for multiply & divide

MIPS arithmetic

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

Example:

C code: A = B + C
MIPS code: add $s0, $s1, $s2

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

MIPS code:
```
add  $t0,  $s1, $s2
add  $s0, $t0,  $s3
sub  $s4, $s5,  $s0
```

MIPS Load-Store Architecture

- Every operand must be in a register (a few exceptions)
- Variables have to be loaded in registers.
- Results have to be stored in memory.

a = b + c
d = a + b

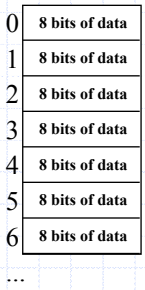
load b in register Rx
load c in register Ry
Rz = Rx + Ry
store Rz in a

Rt = Rz + Rx
store Rt in d

more variables than registers, so need explicit load and stores.

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Q: How to specify a memory location?

Load & Store Instructions

- Base+Offset addressing mode: offset (base register)

  e.g., 32($s3)

- Example:
  - C code: `A[8] = h + A[8];`
    - `A: an array of 100 words`
    - the base address of the array A is in $s3

    base register: $s3
    offset: 32

  - MIPS code:

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 32($s3)
```

- Store word has destination last

Example: Compiling using a Variable Array Index

- C code: g = h + A[i]
  $s3: base register for A
  g, h, i: $s1, $s2, $s4

- MIPS code:

```
add $t1, $s4, $s4        # $t1 = 2 * i
add $t1, $t1, $t1        # $t1 = 4 * i

add $t1, $t1, $s3        # $t1 = address of A[i]
lw $t0, 0($t1)           # $t0 = A[i]

add $s1, $s2, $t0        # g = h + A[i]
```

LUI instruction

- Load upper immediate
- LUI rt, zero-ext-imm
- Shifts 16-bit immediate into high-order 16 bits, with 16 zeros in low order bits -> rt

- How is LUI useful?

Control: bne & beq

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

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:    ....
```

SLT instructions

- Set if less than
- *E.g. SLTI rt, rs, signed-imm*

If (rs < imm), rt=1, else rt=0

What is SLT used for?

Jumps (J, JAL, JR, JALR)

- MIPS unconditional branch instructions:

```
j  label
```

- Example:

```
if (i!=j)        beq $s4, $s5, Lab1
    h=i+j;       add $s3, $s4, $s5
else             j Lab2
    h=i-j;       Lab1:    sub $s3, $s4, $s5
                 Lab2:    ...
```

- PC <- PC + 4 + 4*SEXT(literal)

Jumps

- JAL target (jump and link)
  - PC+8 -> R31
  - Why PC+8?
  - How is JAL useful?

Jumps

- JR rs
  - Jumps to address in register
- PC <- Reg[rs]
- JR vs. J or JAL?

Jumps

- JALR – Jump and link register
- JALR rd, rs
  - Jumps to rs
  - Writes link address into rd
- Why JALR vs. JAL?

MIPS: Stack detective!

- Call procedure: jump and link (`jal`)
- Return from procedure: jump register (`jr`)
- Argument values: $a0 - $a3
- Return value: $v0
- Template:
  - Call setup
  - Prologue
  - Epilogue
  - Return cleanup

Register Name	Software Name (from regdef.h)	Use and Linkage
$0		Always has the value 0.
$at		Reserved for the assembler.
$2..$3	v0-v1	Used for expression evaluations and to hold the integer type function results. Also used to pass the static link when calling nested procedures.
$4..$7	a0-a3	Used to pass the first 4 words of integer type actual arguments, their values are not preserved across procedure calls.
$8..$15	t0-t7	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$16..$23	s0-s7	Saved registers. Their values must be preserved across procedure calls.
$24..$25	t8-t9	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$26..27 or $kt0..$kt1	k0-k1	Reserved for the operating system kernel.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.
$30 or $fp	fp	Contains the frame pointer (if needed); otherwise a saved register (like s0-s7).
$31	ra	Contains the return address and is used for expression evaluation.

```
Example from Prof. David Wood, University of Wisconsin-Madison
# procedure: procA
# input parameters: $a0 and $a1
# output (return value): $v0
# saved registers: $s0, $s1
# temporary registers: $t0, $t1
# local variables: 5 integers named R, S, T, U, V
# procA calls procB with 5 parameters (R, S, T, U, V).
#
# Stack frame layout:
#
#      | in $a1   |  68($sp)
#      | in $a0   |  64($sp)
#      |---------|
#      |    V     |  60($sp)       --|
#      |    U     |  56($sp)         |
#      |    T     |  52($sp)         |
#      |    S     |  48($sp)         |
#      |    R     |  44($sp)         |
#      |   $t1    |  40($sp)         |
#      |   $t0    |  36($sp)         |  --  A's activation record
#      |   $ra    |  32($sp)         |
#      |   $s1    |  28($sp)         |
#      |   $s0    |  24($sp)         |
#      | out arg4 |  20($sp)         |
#      | out $a3  |  16($sp)         |
#      | out $a2  |  12($sp)         |
#      | out $a1  |   8($sp)         |
#      | out $a0  |   4($sp)       --|
#      |---------|
#      |          | <-- $sp        ---
#      |          |                  |
#      |          |                  |   --  where B's activation record
#      |          |                  |         will be
```


Procedure call setup

1. Place current parameters into stack (space already allocated by caller of this procedure)
2. Save any TEMPORARY registers that need to be preserved across the procedure call
3. Place first 4 parameters to procedure into $a0-$a3
4. Place remainder of parameters to procedure into allocated space within the stack frame

Procedure call setup

```
# call setup for call to procB
# save current (live) parameters into the space specifically
# allocated for this purpose within caller's stack frame
sw $a0, 64($sp)         # only needed if values are 'live'
sw $a1, 68($sp)         # only need if values are 'live'
# save any registers that need to be preserved across the call
sw $t0, 36($sp)         # only need if values are 'live'
sw $t1, 40($sp)         # only need if values are 'live'
# put parameters into proper location
lw $a0, 44($sp)         # load R into $a0
lw $a1, 48($sp)         # load S into $a1
lw $a2, 52($sp)         # load T into $a2
lw $a3, 56($sp)         # load U into $a3
lw $t0, 60($sp)         # load V into a temp register
sw $t0, 20($sp)         # outgoing arg4 must go on the stack
#end call setup
```


Prologue

1. allocate space for stack frame
2. save return address in stack frame
3. copy needed parameters from stack frame into registers
4. save any needed SAVED registers into current stack frame

```
procA:
      # procedure prologue
      sub $sp, $sp, 60          #allocate activation record, include
                                # space for maximum outgoing args
      sw $ra, 32($sp)           #save return address
      sw $s0, 24($sp)           # save 'saved' registers to stack
      sw $s1, 28($sp)           # save 'saved' registers to stack
      # end prologue
```

Time to actually call function! ☺

```
# procedure call
jal procB
```


Return cleanup

1. copy needed return values and parameters from $v0-v1, $a0-a3, or stack frame to correct places
2. restore any temporary registers from stack frame (saved in call setup)

```
# return cleanup for call to procB
# restore saved registers
lw $a0, 64($sp)
lw $a1, 68($sp)
lw $t0, 36($sp)
lw $t1, 40($sp)
# return values are in $v0 and $v1
```

Epilogue

1. restore (copy) return address from stack frame into $ra
2. restore from stack frame any saved registers (saved in prologue)
3. de-allocate stack frame (move $sp so the space for the procedure's frame is gone)

```
# procedure epilogue
# restore return address
lw $ra, 32($sp)
# restore $s registers saved in prologue
lw $s0, 24($sp)
lw $s1, 28($sp)
# put return values in $v0 and $v1
mov $v0, $t0
# deallocate stack frame
add $sp, $sp, 60
# return
jr $ra
```


MIPS coprocessor 0 instructions:mfc0, mtc0

- interrupts, exceptions, resets
- Beta vs MIPS

Exception Registers

- Not part of the register file.
  - `Cause`
    - Records the cause of the exception
  - `EPC` (Exception PC)
    - Records the PC where the exception occurred
- `EPC` and `Cause`: part of Coprocessor 0
- Move from Coprocessor 0
  - `mfc0 $t0, EPC`
  - Moves the contents of `EPC` into `$t0`

Exceptions

Save cause and exception PC

Jump to exception handler (0x0000_1100)

Exception handler:

Saves registers on stack

Reads the `Cause` register

```
mfc0 Cause, $t0
```

Handles the exception

Restores registers

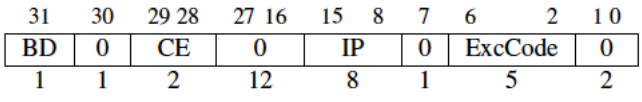
Returns to program

```
mfc0 EPC, $k0
jr $k0
```

Exception Causes

ExcCode	Mnemonic	Description
0	Hint	External interrupt.
2	Tint	Timer interrupt.
4	AdEL	Address or misalignment error on load.
5	AdES	Address or misalignment error on store.
6	AdEF	Address or misalignment error on fetch.
8	Sys	Syscall exception.
9	Bp	Breakpoint exception.
10	RI	Reserved instruction exception.
11	CpU	Coprocessor Unusable.
12	Ov	Arithmetic Overflow.

Cause register



Reset

- mtc0 zero, $9 #init counter
- mtc0 zero, $11 #timer interrupt
- :
- :
- J kernel_init