Computer Architecture and Operating System

Prof. Indranil Sengupta

Department of Computer Science and Engineering

IIT Kharagpur

RISC and CISC Architecture

Broad Classification

- Computer architectures have evolved over the years.
  - Features that were developed for mainframes and supercomputers in the 1960s and 1970s have started to appear on a regular basis on later generation microprocessors.
- Two broad classifications of ISA:
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)

CISC versus RISC Architectures

- **Complex Instruction Set Computer (CISC)**
  - More traditional approach.
  - Main features:
    - Complex instruction set
    - Large number of addressing modes (R-R, R-M, M-M, indexed, indirect, etc.)
    - Special-purpose registers and Flags (sign, zero, carry, overflow, etc.)
    - Variable-length instructions / Complex instruction encoding
    - Ease of mapping high-level language statements to machine instructions
    - Instruction decoding / control unit design more complex
    - Pipeline implementation quite complex

- CISC Examples:
  - IBM 360/370 (1960-70)
  - VAX-11/780 (1970-80)
  - Intel x86 / Pentium (1985-present)

Only CISC instruction set that survived over generations.
- Desktop PC's / Laptops use these.
- The volume of chips manufactured is so high that there is enough motivation to pay the extra design cost.
- Sufficient hardware resources available today to translate from CISC to RISC internally.

- **Reduced Instruction Set Computer (RISC)**
  - Very widely used among many manufacturers today.
  - Also referred to as *Load-Store Architecture*.
    - Only LOAD and STORE instructions access memory.
    - All other instructions operate on processor registers.
  - Main features:
    - Simple architecture for the sake of efficient pipelining.
    - Simple instruction set with very few addressing modes.
    - Large number of general-purpose registers; very few special-purpose.
    - Instruction length and encoding uniform for easy instruction decoding.
    - Compiler assisted scheduling of pipeline for improved performance.

- RISC Examples:
  - CDC 6600 (1964)
  - MIPS family (1980-90)
  - SPARC
  - ARM microcontroller family
  - RISC-V

- Almost all the computers today use a RISC based pipeline for efficient implementation.
  - RISC based computers use compilers to translate into RISC instructions.
  - CISC based computers (e.g. x86) use hardware to translate into RISC instructions.

Results of a Comparative Study

- A quantitative comparison of VAX 8700 (a CISC machine) and MIPS M2000 (a RISC machine) with comparable organizations was carried out in 1991.
- Some findings:
  - MIPS required execution of about twice the number of instructions as compared to VAX.
  - Cycles Per Instructions (CPI) for VAX was about six times larger than that of MIPS.
  - Hence, MIPS had three times the performance of VAX.
  - Also, much less hardware is required to build MIPS as compared to VAX.

MIPS32 Architecture: A Case Study

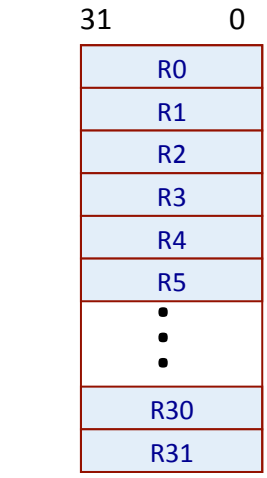
MIPS32 Architecture

- As a case study of RISC ISA, we shall be considering the MIPS32 architecture.
  - Look into the instruction set and instruction encoding in detail.
  - Design the data path of the MIPS32 architecture, and also look into the control unit design issues.
  - Extend the basic data path of MIPS32 to a pipeline architecture, and discuss some of the issues therein.

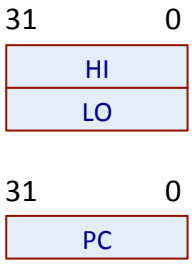
MIPS32 CPU Registers

- The MIPS32 ISA defines the following CPU registers that are visible to the machine/assembly language programmer.
  a) 32, 32-bit general purpose registers (GPRs), R0 to R31.
  b) A special-purpose 32-bit program counter (PC).
     - Points to the next instruction in memory to be fetched and executed.
     - Not directly visible to the programmer.
     - Affected only indirectly by certain instructions (like branch, call, etc.)
  c) A pair of 32-bit special-purpose registers HI and LO, which are used to hold the results of multiply, divide, and multiply-accumulate instructions.

- Some common registers are missing in MIPS32.
  - ***Stack Pointer*** (SP) register, which helps in maintaining a stack in main memory.
    - Any of the GPRs can be used as the stack pointer.
    - No separate PUSH, POP, CALL and RET instructions.
  - ***Index Register*** (IX), which helps in accessing memory words sequentially in memory.
    - Any of the GPRs can be used as an index register.
  - ***Flag registers*** (like ZERO, SIGN, CARRY, OVERFLOW) that keeps track of the results of arithmetic and logical operations.
    - Maintains flags in registers, to avoid problems in pipeline implementation.



General Purpose Registers



Special Purpose Registers

Two of the GPRs have assigned functions:

a) R0 is hard-wired to a value of zero.
- Can be used as the target register for any instruction whose result is to be discarded.
- Can also be used as a source when a zero value is needed.

b) R31 is used to store the return address when a function call is made.
- Used by the jump-and-link and branch-and-link instructions like JAL, BLTZAL, BGEZAL, etc.
- Can also be used as a normal register.

Some Examples

```
LD      R4, 50(R3)      // R4 = Mem[50+R3]
ADD     R2, R1, R4      // R2 = R1 + R4
SD      54(R3), R2      // Mem[54+R3] = R2
```

```
ADD   R2, R5, R0      // R2 = R5
```

```
MAIN:  ADDI   R1, R0, 35    // R1 = 35
       ADDI   R2, R0, 56    // R2 = 56
       JAL    GCD
       .....

GCD:   .....      // Find GCD of R1 & R2

       JR     R31
```

How are the HI and LO registers used?

- During a multiply operation, the HI and LO registers store the product of an integer multiply.
  - HI denotes the high-order 32 bits, and LO denotes the low-order 32 bits.
- During a multiply-add or multiply-subtract operation, the HI and LO registers store the result of the integer multiply-add or multiply-subtract.
- During a division, the HI and LO registers store the quotient (in LO) and remainder (in HI) of integer divide.

Some MIPS32 Assembly Language Conventions

- The integer registers of MIPS32 can be accessed as R0..R31 or r0..r31 in an assembly language program.
- Several assemblers and simulators are available in the public domain (like QtSPIM) that follow some specific conventions.
  - These conventions have become like a *de facto* standard when we write assembly language programs for MIPS32.
  - Basically some alternate names are used for the registers to indicate their intended usage.

Register name	Register number	Usage
$zero	R0	Constant zero

Used to represent the constant zero value, wherever required in a program.

Register name	Register number	Usage
$at	R1	Reserved for assembler

May be used as temporary register during macro expansion by assembler.

- Assembler provides an extension to the MIPS32 instruction set that are converted to standard MIPS32 instructions.

Example: Load Address instruction used to initialize pointers

```
la    R5,addr
      ⇩
lui   $at,Upper-16-bits-of-addr
ori   R5,$at,Lower-16-bits-of-addr
```

Register name	Register number	Usage
$v0	R2	Result of function, or for expression evaluation
$v1	R3	Result of function, or for expression evaluation

May be used for up to two function return values, and also as temporary registers during expression evaluation.

Register name	Register number	Usage
$a0	R4	Argument 1
$a1	R5	Argument 2
$a2	R6	Argument 3
$a3	R7	Argument 3

May be used to pass up to four arguments to functions.

Register name	Register number	Usage
$t0	R8	Temporary (not preserved across call)
$t1	R9	Temporary (not preserved across call)
$t2	R10	Temporary (not preserved across call)
$t3	R11	Temporary (not preserved across call)
$t4	R12	
$t5	R13	
$t6	R14	
$t7	R15	
$t8	R24	Temporary (not preserved across call)
$t9	R25	Temporary (not preserved across call)

May be used as temporary variables in programs.

These registers might get modified when some functions are called (other than user-written functions).

Register name	Register number	Usage
$s0	R16	Temporary (preserved across call)
$s1	R17	Temporary (preserved across call)
$s2	R18	Temporary (preserved across call)
$s3	R19	Temporary (preserved across call)
$s4	R20	Temporary (preserved across call)
$s5	R21	Temporary (preserved across call)
$s6	R22	
$s7	R23	

May be used as temporary variables in programs.

These registers do not get modified across function calls.

Register name	Register number	Usage
$gp	R28	Pointer to global area
$sp	R29	Stack pointer
$fp	R30	Frame pointer
$ra	R31	Return address (used by function call)

These registers are used for a variety of pointers:

- Global area: points to the memory address from where the global variables are allocated space.
- Stack pointer: points to the top of the stack in memory.
- Frame pointer: points to the activation record in stack.
- Return address: used while returning from a function.

Register name	Register number	Usage
$k0	R26	Reserved for OS kernel
$k1	R27	Reserved for OS kernel

These registers are supposed to be used by the OS kernel in a real computer system.

It is highly recommended not to use these registers.