U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Instructions (Chapter 2)

www.cs.wisc.edu/~karu/courses/cs552/

Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar,Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Instruction Set Architecture (ISA)

- The **"contract"** between software and hardware
  - **Functional definition** of operations, modes, and storage locations supported by hardware
  - **Precise description** of how software can invoke and access them
- Strictly speaking, ISA is the architecture
  - Informally, architecture is also used to talk about the big picture of implementation
  - Better to call this **micro-architecture**

Microarchitecture

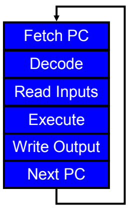
- ISA specifies what hardware does, not how it does it
  - No guarantees regarding
    - How operations are implemented
    - Which operations are fast and which are slow
    - Which operations take more power and which take less
  - These issues are determined by the microarchitecture
    - Microarchitecture = how hardware implements architecture
    - All Pentiums implement the x86 architecture

Aspects of ISAs

1. The Von Neumann model
   - Implicit structure of all modern ISAs
2. Format
   - Length and encoding
3. Operations
4. Operand model
   - Where are operands stored and how do address them?
5. Datatypes and operations
6. Control

- Running example: MIPS
- Your project will use 16-bit MIPS-lite
- Touch on x86

(1) The Sequential (Von Neumann) Model



- Implicit model of all modern ISAs
  -
- Key: **program counter (PC)**
  - Defines **total order** of dynamic instructions
    - Next PC is PC++ unless insn says otherwise
  - Order and **named storage** define computation
    - Value flows from insn X to Y via storage A iff…
    - X names A as output, Y names A as input…
    - And Y after X in total order
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts

(2) Instruction Format

- **Length**
  1. Fixed length
     - 32 or 64 bits (your project: 16 bit ISA)
     - \+ Simple implementation: compute next PC using only PC
     - – Code density
  2. Variable length
     - – Complex implementation
     - \+ Code density
  3. Compromise: two lengths
     - Example: $MIPS_{16}$
- **Encoding**
  - A few simple encodings simplify decoder implementation
  - Complex encoding can improve code density

MIPS Format

- Length
  - 32-bits
  - $MIPS_{16}$: 16-bit variants of common instructions for density
- Encoding
  - 3 formats, simple encoding
  - Q: how many operation types can be encoded in 6-bit opcode?

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)
I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)		
J-type	Op(6)	Target(26)				

R Format

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- E.g., add $1, $2, $3

000000	00010	00011	00001	00000	100000
alu-rr	2	3	1	zero	add/signed

How do you store the number 4,392,992?

.

I Format

- All loads and stores use I-format
- Assembly: lw $1, 100($2)
- Machine:

100011 00010 00001 0000000001100100
lw 2 1 100 (in binary)

opcode	rs	rt	addr/immediate
6	5	5	16

I Format, cont.

- ALU ops with immediates
  - addi $1, $2, 100
  - 001000 00010 00001 0000000001100100
- Conditional branches
  - beq $1, $2, 7
  - 000100 00001 00010 0000 0000 0000 0111
  - PC = PC + (0000 0111 << 2) // word offset

J Format

Weird Direct Jump:

opcode	addr
6	26

- Jump to:
  - New PC = 4 MSB of PC || addr || 00
  - 4+26+2 = 32 bits for jump target

(3) Operations

- Operation type encoded in instruction opcode
- Many types of operations
  - Integer arithmetic: add, sub, mul, div, mod/rem (signed/unsigned)
  - FP arithmetic: add, sub, mul, div, sqrt
  - Integer logical: and, or, xor, not, sll, srl, sra
  - Packed integer: padd, pmul, pand, por… (saturating/wraparound)
- What other operations might be useful?
- More operation types == better ISA??
- DEC VAX computer had LOTS of operation types
  - E.g., instruction for polynomial evaluation (no joke!)
  - But many of them were rarely/never used

(4) Operations Act on Operands

- If you're going to add, you need at least 3 operands
  - Two source operands, one destination operand
- Question #1: Where can operands come from?
- Question #2: And how are they specified?

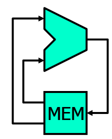
- Running example: A = B + C
  - Several options for answering both questions

- Discuss: Memory-Only & Registers
- Optional: Accumulator & Stack

Operand Model I: Memory Only

- **Memory only**

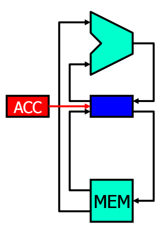
`add A,B,C`	mem[A] = mem[B] + mem[C]



Operand Model II: Accumulator

- **Accumulator**: implicit single-element stack

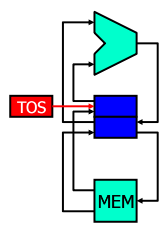
`load B`	ACC = mem[B]
`add C`	ACC = ACC + mem[C]
`store A`	mem[A] = ACC



Operand Model III: Stack

- **Stack**: top of stack (TOS) is implicit in instructions

`push B`	stack[TOS++] = mem[B]
`push C`	stack[TOS++] = mem[C]
`add`	stack[TOS++] = stack[--TOS] + stack[--TOS]
`pop A`	mem[A] = stack[--TOS]



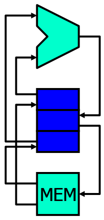
Operand Model: Registers

- **General-purpose registers**: multiple explicit accumulators

`load R1,B`	R1 = mem[B]
`add R1, C`	R1 = R1 + mem[C]
`store R1,A`	mem[A] = R1

- **Load-store**: GPR and only loads/stores access memory

`load R1,B`	R1 = mem[B]
`load R2,C`	R2 = mem[C]
`add R1,R1,R2`	R1 = R1 + R2
`store R1,A`	mem[A] = R1



Operand Model Pros and Cons

- Metric I: **static code size**
  - Number of instructions needed to represent program, size of each
  - Evaluation: register < load-store < memory only
- Metric II: **data memory traffic**
  - Number of bytes moved to and from memory
  - Evaluation: load-store < register < memory only
- Metric III: **instruction latency**
  - Want low latency to execute instructions
  - Evaluation: load-store < register < memory only
- Upshot: most current ISAs are load-store

MIPS Operand Model

- MIPS is load-store
  - 32 32-bit integer registers
    - Actually 31: r0 is hardwired to value 0 → why?
  - 32 32-bit FP registers
    - Can also be treated as 16 64-bit FP registers
  - HI,LO: destination registers for multiply/divide
- Integer register conventions
  - Allows separate function-level compilation and fast function calls

Memory Addressing

- ISAs assume **"virtual" address size**
  - Either 32 or 64 bits
  - Program can name 2^{32} bytes (4GB) or 2^{64} bytes (16PB)
  - ISA point? no room for even one address in a 32-bit instruction
- **Addressing mode**: way of specifying address
  - **Direct:** `ld R1,(R2)` R1=mem[R2]
  - **Displacement:** `ld R1,8(R2)` R1=mem[R2+8]
  - **Indexed:** `ld R1,(R2,R3)` R1=mem[R2+R3]
  - **Memory-indirect:** `ld R1,@(R2)` R1=mem[mem[R2]]
  - **Auto-update:** `ld R1,8(R2)` R2+=8; R1=mem[R2]
  - **Scaled:** `ld R1,(R2,R3,32,8)` R1=mem[R2+R3*32+8]
  - What high-level program idioms are these used for?

MIPS Addressing Modes

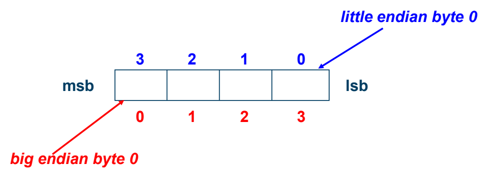
- MIPS implements only displacement
  - Why? Experiment on VAX (ISA with every mode) found distribution
  - Disp: 61%, reg-ind: 19%, scaled: 11%, mem-ind: 5%, other: 4%
  - 80% use displacement or register indirect (=displacement 0)
- I-type instructions: 16-bit displacement
  - Is 16-bits enough?
  - Yes! VAX experiment showed 1% accesses use displacement >16

I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)

Addressing Issue: Endian-ness

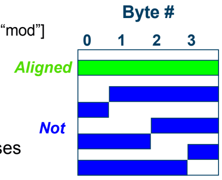
Byte Order

- Big Endian: byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, SPARC, HP PA-RISC
- Little Endian: byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC/Compaq Alpha



Another Addressing Issue: Alignment

- Alignment: require that objects fall on address that is multiple of their size
- 32-bit integer
  - Aligned if address % 4 = 0 [% is symbol for "mod"]
  - Aligned: `lw @XXXX00`
  - Not: `lw @XXXX10`
- 64-bit integer?
  - Aligned if ?
- Question: what to do with unaligned accesses (uncommon case)?
  - Support in hardware? Makes all accesses slow
  - Trap to software routine? Possibility
  - **MIPS? ISA support**: unaligned access using two instructions:
    `lw @XXXX10 = lwl @XXXX10; lwr @XXXX10`



(5) Datatypes

- Datatypes
  - Software view: property of data
  - Hardware view: data is just bits, property of operations
- Hardware datatypes
  - Integer: 8 bits (byte), 16b (half), 32b (word), 64b (long)
  - IEEE754 FP: 32b (single-precision), 64b (double-precision)
  - Packed integer: treat 64b int as 8 8b int's or 4 16b int's

MIPS Datatypes (and Operations)

- Datatypes: all the basic ones (byte, half, word, FP)
  - All integer operations read/write 32-bits
    - No partial dependences on registers
  - Only byte/half variants are load-store
    `lb, lbu, lh, lhu, sb, sh`
  - Loads sign-extend (or not) byte/half into 32-bits
- Operations: all the basic ones
  - Signed/unsigned variants for integer arithmetic
  - Immediate variants for all instructions
    `add, addu, addi, addiu`
  - **Regularity/orthogonality**: all variants available for all operations
    - Makes compiler's "life" easier

(6) Control Instructions I

- One issue: **testing for conditions**
  - Option I: **compare and branch instructions**
    `blti $1,10,target`
    + Simple, – two ALUs: one for condition, one for target address
  - Option II: **implicit condition codes**
    `subi $2,$1,10   // sets "negative" CC`
    `bn target`
    + Condition codes set "for free", – implicit dependence is tricky
  - Option III: **condition registers, separate branch insns**
    `slti $2,$1,10`
    `bnez $2,target`
    – Additional instructions, + one ALU per, + explicit dependence

MIPS Conditional Branches

- MIPS uses combination of options II and III
  - Compare 2 registers and branch: `beq`, `bne`
    - Equality and inequality only
    + Don't need an adder for comparison
  - Compare 1 register to zero and branch: `bgtz`, `bgez`, `bltz`, `blez`
    - Greater/less than comparisons
    + Don't need adder for comparison
  - Set explicit condition registers: `slt`, `sltu`, `slti`, `sltiu`, etc.
- Why? 86% of branches in programs are (in)equalities or comparisons to 0

Control Instructions II

- Another issue: **computing targets**
  - Option I: **PC-relative**
    - Position-independent within procedure
    - Used for branches and jumps within a procedure
  - Option II: **Absolute**
    - Position independent outside procedure
    - Used for procedure calls
  - Option III: **Indirect** (target found in register)
    - Needed for jumping to dynamic targets
    - Used for returns, dynamic procedure calls, switches
  - How far do you need to jump?
    - Typically not so far within a procedure (they don't get that big)
    - Further from one procedure to another

MIPS Control Instructions

- MIPS uses all three
  - PC-relative → conditional branches: `bne`, `beq`, `blez`, etc.
    - 16-bit relative offset, <0.1% branches need more
    - PC = PC + 4 + immediate if condition is true (else PC=PC+4)

I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)

  - Absolute → unconditional jumps: `j target`
    - 26-bit offset (can address 2^{28} words < 2^{32} → what gives?)

J-type	Op(6)	Target(26)

  - Indirect → Indirect jumps: `jr $rd`

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)

Control Instructions III

- Another issue: support for procedure calls?
  - We "link" (remember) address of the calling instruction + 4 (current PC + 4) so we can return to it after the procedure

- MIPS
  - Implicit return address register is `$ra(=$31)`
  - Direct jump-and-link: `jal address`
    → $ra = PC+4; PC = address
  - Can then return from call with: `jr $ra`
  - Or can call with indirect jump-and-link: `jalr $rd, $rs`
    → $rd = PC+4; PC = $rs // explicit return address register
  - Then return with: `jr $rd`

Control Idiom: If-Then-Else

- Understanding programs helps with architecture
  - Know what common programming idioms look like in assembly
  - Why? How can you MCCF if you don't know what CC is?

- First control idiom: **if-then-else**

```
if (A < B) A++;      // A in $s1
else B++;            // B in $s2

        slt  $s3,$s1,$s2     // if $s1<$s2, then $s3=1
        beqz $s3,else        // branch to else if !condition
        addi $s1,$s1,1
        j    join            // jump to join
else:   addi $s2,$s2,1
join:
```


Control Idiom: Arithmetic For Loop

- Second idiom: **for loop with arithmetic induction**

```
int A[100], sum, i, N;
for (i=0; i<N; i++){          // assume: i in $s1, N in $s2
  sum += A[i];                // &A[i] in $s3, sum in $s4
}
        sub  $s1,$s1,$s1      // initialize i to 0
loop:   slt  $t1,$s1,$s2      // if i<N, then $t1=1
        beqz $t1,exit         // test for exit at loop header
        lw   $t1,0($s3)       // $t1 = A[i]  (not &A[i])
        add  $s4,$s4,$t1      // sum = sum + A[i]
        addi $s3,$s3,4        // increment &A[i] by sizeof(int)
        addi $s1,$s1,1        // i++
        j loop                // backward jump
exit:
```

Outline

- Instruction Sets in General
- MIPS Assembly Programming
- Other Instruction Sets
  - Goals of ISA Design
  - RISC vs. CISC
  - Intel x86 (IA-32)

RISC vs. CISC

- **RISC**: reduced-instruction set computer
  - Coined by P+H in early 80's
- **CISC**: complex-instruction set computer
  - Not coined by anyone, term didn't exist before "RISC"
- Religious war (one of several) started in mid 1980's
  - RISC (MIPS, Alpha) "won" the technology battles
  - CISC (IA32 = x86) "won" the commercial war
    - Compatibility a stronger force than anyone (but Intel) thought
    - Intel beat RISC at its own game … more on this soon

Intel 80x86 ISA (aka x86 or IA-32 now)

- Long history
- Binary compatibility across generations
- 1978: 8086, 16-bit, registers have dedicated uses
- 1980: 8087, added floating point (stack)
- 1982: 80286, 24-bit
- 1985: 80386, 32-bit, new instrs → GPR almost
- 1989-95: 80486, Pentium, Pentium II
- 1997: Added MMX instructions (for graphics)
- 1999: Pentium III
- 2002: Pentium 4
- 2004: "Nocona" 64-bit extension (to keep up with AMD)

Intel x86: The Penultimate CISC (VAX ultimate)

- Variable length instructions: 1-16 bytes
- Few registers: 8 and each one has a special purpose
- Multiple register sizes: 8,16,32 bit (for backward compatibility)
- Accumulators for integer instrs, and stack for FP instrs
- Multiple addressing modes: indirect, scaled, displacement
- Register-register, memory-register, and memory-register insns
- Condition codes
- Instructions for memory stack management (push, pop)
- Instructions for manipulating strings (entire loop in one instruction)

- Summary: yuck!

80x86 Registers and Addressing Modes

- Eight 32-bit registers (not truly general purpose)
  - EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- Six 16-bit Registers for code, stack, & data
- 2-address ISA
  - One operand is both source and destination
- NOT a Load/Store ISA
  - One operand can be in memory

80x86 Addressing Modes

- Register Indirect
  - mem[reg]
  - not ESP or EBP
- Base + displacement (8 or 32 bit)
  - mem[reg + const]
  - not ESP or EBP
- Base + scaled index
  - mem[reg + (2^{scale} x index)]
  - scale = 0,1,2,3
  - base any GPR, index not ESP
- Base + scaled index + displacement
  - mem[reg + (2^{scale} x index) + displacement]
  - scale = 0,1,2,3
  - base any GPR, index not ESP

Condition Codes

- x86 ISA has condition codes
- Special HW register that has values set as side effect of instruction execution
- Example conditions
  - Zero
  - Negative
- Example use

```
subi $t0, $t0, 1
bz  loop
```


80x86 Instruction Encoding

- Variable size 1-byte to 17-bytes
- Jump (JE) 2-bytes
- Push 1-byte
- Add Immediate 5-bytes
- W bit says 32-bits or 8-bits
- D bit indicates direction
  - memory → reg or reg → memory
  - movw EBX, [EDI + 45]
  - movw [EDI + 45], EBX

Decoding x86 Instructions

- Is a nightmare!
- Instruction length is variable from 1 to 17 bytes!
- Prefixes, postfixes
- Crazy “formats” → register specifiers move around
- But key instructions not terrible
- Yet, everything must work correctly

How x86 Won Anyway

- X86 won because it was the first 16-bit chip by 2 years
- IBM put it into its PCs because no competing choice
- Software written to x86 so x86 is the standard
- Hard to complete with Intel
  - X86 is difficult ISA to implement
  - Intel can amortize design effort over vast sales
  - Intel uses RISC “underneath”
- Moore’s law has helped in a big way
  - Most engineering problems can be solved with more transistors