U. Wisconsin CS/ECE 752 Advanced Computer Architecture I

Prof. David A. Wood

Unit 2: Instruction Set Architecture

Slides developed by Amir Roth of University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

Slides enhanced by Milo Martin, Mark Hill, and David Wood with sources that included Profs. Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Sohi, Vijaykumar, and Wood

What Is An ISA?

- ISA (instruction set architecture)
  - A well-define hardware/software interface
  - The **"contract"** between software and hardware
    - **Functional definition** of operations, modes, and storage locations supported by hardware
    - **Precise description** of how to invoke, and access them
  - No guarantees regarding
    - How operations are implemented
    - Which operations are fast and which are slow and when
    - Which operations take more power and which take less

A Language Analogy for ISAs

- A ISA is analogous to a human language
  - Allows communication
    - Language: person to person
    - ISA: hardware to software
  - Need to speak the same language/ISA
  - Many common aspects
    - Part of speech: verbs, nouns, adjectives, adverbs, etc.
    - Common operations: calculation, control/branch, memory
  - Many different languages/ISAs, many similarities, many differences
    - Different structure
  - Both evolve over time
- Key differences: ISAs must be unambiguous
  - ISAs are explicitly engineered and extended

RISC vs CISC Foreshadowing

- Recall performance equation:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)
- CISC (Complex Instruction Set Computing)
  - Improve "instructions/program" with "complex" instructions
  - Easy for assembly-level programmers, good code density
- RISC (Reduced Instruction Set Computing)
  - Improve "cycles/instruction" with many single-cycle instructions
  - Increases "instruction/program", but hopefully not as much
    - Help from smart compiler
  - Perhaps improve clock cycle time (seconds/cycle)
    - via aggressive implementation allowed by simpler instructions

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Implementability**
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design high-reliability implementations?
    - Easy to design low-cost implementations?
- **Compatibility**
  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4,…

Programmability

- Easy to express programs efficiently?
  - For whom?
- Early: **human**
  - Compilers were terrible, most code was hand-assembled
  - Want high-level coarse-grain instructions
    - As similar to high-level language as possible
- Last decades: **compiler**
  - Optimizing compilers usually generate better code than you or I
  - Want low-level fine-grain instructions
    - Compiler can't tell if two high-level idioms match exactly or not

Human Programmability

- What makes an ISA easy for a human to program in?
  - Proximity to a high-level language (HLL)
    - Closing the "**semantic gap**"
  - Semantically heavy (CISC-like) insns that capture complete idioms
    - "Access array element", "loop", "procedure call"
    - Example: SPARC `save/restore`
    - Bad example: x86 `rep movsb` (copy string)
    - Ridiculous example: VAX `insque` (insert-into-queue)
  - "**Semantic clash**": what if you have many high-level languages?
- Stranger than fiction
  - People once thought computers would execute language directly
  - Fortunately, never really happened (except Symbol)

Compilers 101

- Compiler goals:
  - all correct programs execute correctly
  - most compiled programs execute fast
  - compile fast
  - provide support for debugging
- Use multiple phases to manage complexity
  - Lexical analysis (e.g., "+" means "add", "foobar" is an identifier)
  - Parsing (e.g., "x = a + b" means assign sum of variables a and b to x)
    - Generates intermediate representation
  - Optimization & code generation (transforms intermediate representation)
    - Procedure In-lining, Loop optimizations, Common sub-expression elimination, Jump optimization, Constant propagation, Register allocation, Strength reduction, Pipeline scheduling, Interprocedural analysis
  - Generation of assembly code

Which comes first? Phase ordering problem.

Compiler Programmability

- What makes an ISA easy for a compiler to program in?
  - Low level primitives from which solutions can be synthesized
    - Wulf: "**primitives not solutions**"
    - Compilers good at breaking complex structures to simple ones
      - Requires decomposition
    - Not so good at combining simple structures into complex ones
      - Requires search, pattern matching (why AI is hard)
    - Easier to synthesize complex insns than to compare them
  - Rules of thumb
    - Regularity: "**principle of least astonishment**"
    - Orthogonality & composability
    - One-vs.-all

Implementability

- Every ISA can be implemented
  - Not every ISA can be implemented efficiently (at least easily)
- Classic high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution (more later)
- Certain ISA features make these difficult
  - Variable instruction lengths/formats: complicate decoding
  - Implicit state: complicates dynamic scheduling
  - Variable latencies: complicates scheduling
  - Difficult to interrupt instructions: complicate many things
  - A solution: High-performance x86 machines dynamically translate CISC instructions into internal micro-ops (e.g., RISC-ops)

Compatibility

- No-one buys new hardware… if it requires new software
  - IBM did this for mainframes; Intel for PCs
  - ISA must remain compatible, no matter what
    - x86 arguably one of the worst ISAs EVER, but survives
    - As does IBM's 360/370/390 (the *first* "ISA family")
- **Backward compatibility**
  - New processors must support old programs
    - Can't drop features, but can deprecate and emulate
  - Very important
- **Forward (upward) compatibility**
  - Old processors must support new programs (with software help)
    - New processors redefine only previously-illegal opcodes
    - Allow software to detect support for specific new instructions
    - Old processors emulate new instructions in low-level software

The Compatibility Trap

- Easy compatibility requires forethought
  - Temptation: use some ISA extension for 5% performance gain
  - Frequent outcome: gain diminishes, disappears, or turns to loss
    - Must continue to support gadget for eternity
  - Example: register windows (SPARC)
    - Reduces register spills and fills
    - Adds cost and complexity to out-of-order implementations of SPARC
  - Example: branch delay slot (most RISCs)
    - Eliminates branch hazard in simple 5-stage pipeline
    - Complicates multi-instruction issue (superscalar)

The Compatibility Trap Door

- Compatibility's friends
  - **Trap**: instruction makes low-level "function call" to OS handler
  - **Nop**: "no operation" - instructions with no functional semantics
- Backward compatibility
  - Handle rarely used but hard to implement "legacy" opcodes
  - Define to trap in new implementation and emulate in software
    - Rid yourself of some ISA mistakes of the past
    - Problem: performance suffers for legacy codes
- Forward compatibility
  - Reserve sets of trap & nop opcodes (don't define uses)
  - Add ISA functionality by overloading traps
    - Release firmware patch to "add" to old implementation
  - Add ISA hints by overloading nops

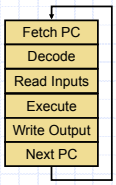
Blocking the Compatibility Trap Door

- Temptation:
  - Define "unused" instruction fields as "don't cares"
    - E.g., MIPS "shift length" field in an "add" instruction
  - Simplifies hardware logic needed to decode instructions
- Trap:
  - Can't use "unused" values for new instructions
  - Same problem for special registers (e.g., Interrupt status register)
- Solution:
  - Define all bits (usually to be zero).

Aspects of ISAs

- **Von Neumann model**
  - Implicit structure of most ISAs
- Format
  - Length and encoding
- **Operand model**
  - Where (other than memory) are operands stored?
- Datatypes and operations
- Control
- Overview only
  - Read about the rest in the book and appendices
  - You MUST be comfortable with MIPS ISA

The Sequential Model



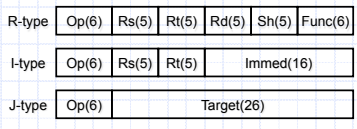
- Implicit model of all modern commercial ISAs
  - Called von Neuman, but in ENIAC design before
- Basic feature: the **program counter (PC)**
  - Defines **total order** on dynamic instruction
    - Next PC is PC++ unless insn says otherwise
  - Order and **named storage** define computation
    - Value 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
- Alternatives have been proposed…

Format

- **Length**
  - Fixed length
    - Most common is 32 bits
    - + Simple implementation: compute next PC using only PC
    - – Code density: 32 bits to increment a register by 1?
      - – x86 can do this in one 8-bit instruction
  - Variable length
    - – Complex implementation
    - + Code density
  - Compromise: two lengths
    - MIPS16 or ARM's Thumb
- **Encoding**
  - A few simple encodings simplify decoder implementation

Example: MIPS Format

- Length
  - 32-bits
- Encoding
  - 3 formats, simple encoding
  - Q: how many instructions can be encoded? A: 64? 127? 4096?

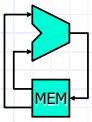


Operand Model: Memory Only

- Where (other than memory) can operands come from?
  - And how are they specified?
  - Example: A = B + C
  - Several options
- **Memory only**

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

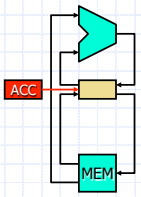
  - Not practical



Operand Model: Accumulator

- **Accumulator**: implicit single element storage

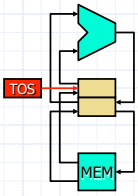
```
load B           ACC = mem[B]
add C            ACC = ACC + mem[C]
store A          mem[A] = ACC
```



Operand Model: Stack

- **Stack**: TOS implicit in instructions

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



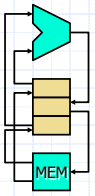
Operand Model: Registers

- **General-purpose register**: multiple explicit accumulator

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

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

```
load B,R1        R1 = mem[B]
load C,R2        R2 = mem[C]
add R1,R2,R1     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
  - Want many implicit operands, high level instructions
  - Good → bad: accumulator, stack, GP-register, load-store
- Metric II: **data memory traffic**
  - Number of bytes move to and from memory
  - Want as many long-lived operands in on-chip storage
  - Good → bad: load-store / GP-register, stack, accumulator,
- Metric III: **cycles per instruction**
  - Want short (1 cycle?), little variability, few nearby dependences
  - Good → bad: load-store, GP-register, stack, accumulator
- Upshot: most new ISAs are load-store (or GP-register)
- Question: Any recent stack architectures?

How Many Registers?

- Registers faster than memory, have as many as possible?
  - **No**
  - One reason registers are faster is that there are **fewer of them**
    - Small is fast (Speed of light, diffusion equation, etc.)
  - Another is that they are **directly addressed** (no address calc)
    - More of them, means larger specifiers
    - Fewer registers per instruction or indirect addressing
  - **Not everything can be put in registers**
    - Structures, arrays, anything pointed-to
    - Although compilers are getting better at putting more things in
  - More registers means **more saving/restoring**
- Upshot: trend to more registers: 8 (x86)→32 (MIPS) →128 (IA64)
  - 64-bit x86 has 16 64-bit integer and 16 128-bit FP registers

Register Windows

- **Register windows**: hardware activation records
  - Sun SPARC (from the RISC I)
  - 32 integer registers divided into: 8 global, 8 local, 8 input, 8 output
  - Explicit `save`/`restore` instructions
    - Global registers fixed
    - `save`: inputs "pushed", outputs → inputs, locals zeroed
    - `restore`: locals zeroed, inputs → outputs, inputs "popped"
    - Hardware stack provides few (8) on-chip register frames
    - Spilled-to/filled-from memory on over/under flow
  - \+ Automatic parameter passing, caller-saved registers
  - \+ No memory traffic on shallow (<8 deep) call graphs
  - – Hidden memory operations (some restores fast, others slow)
  - – A nightmare for register renaming (more later)

Virtual Address Size

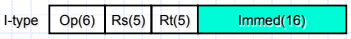
- What is an n-bit processor?
  - Support memory size of 2^n
  - Alternative (wrong) definition: size of calculation operations
- Virtual address size
  - Determines maximum size of addressable (usable) memory
    - Current 32-bit or 64-bit address spaces
    - All ISAs moving to (if not already at) 64 bits
      - Most implementations limited to 40-50 bits
    - A pain to overcome too-small virtual address space
  - x86 evolution:
    - 12-bit (4004), 14-bit (8008), 16-bit (8086), 24-bit (80286),
    - 32-bit + protected memory (80386)
    - 64-bit (AMD's Opteron & Intel's EM64T Pentium4)

Memory Addressing

- **Addressing mode:** way of specifying address
  - Used in memory-memory or load/store instructions in register ISA
- Examples
  - **Register-Indirect:** R1=mem[R2]
  - **Displacement:** R1=mem[R2+immed]
  - **Index-base:** R1=mem[R2+R3]
  - **Memory-indirect:** R1=mem[mem[R2]]
  - **Auto-increment:** R1=mem[R2], R2= R2+1
  - **Auto-indexing:** R1=mem[R2+immed], R2=R2+immed
  - **Scaled:** R1=mem[R2+R3*immed1+immed2]
  - **PC-relative:** R1=mem[PC+imm]
- What high-level program idioms are these used for?

Example: 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 small 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



- SPARC adds Reg+Reg mode

Two More Addressing Issues

- **Access alignment**: address % size == 0?
  - Aligned: `load-word @XXXX00, load-half @XXXXX0`
  - Unaligned: `load-word @XXXX10, load-half @XXXXX1`
  - Question: what to do with unaligned accesses (uncommon case)?
    - Support in hardware? Makes all accesses slow
    - Trap to software routine? Possibility
    - Use regular instructions
      - Load, shift, load, shift, and
    - **MIPS? ISA support**: unaligned access using two instructions

```
lwl @XXXX10; lwr @XXXX10
```

- **Endian-ness**: arrangement of bytes in a word
  - Big-endian: sensible order (e.g., MIPS, PowerPC)
    - A 4-byte integer: "00000000 00000000 00000010 00000011" is 515
  - Little-endian: reverse order (e.g., x86)
    - A 4-byte integer: "00000011 00000010 00000000 00000000 " is 515
  - Why little endian? To be different? To be annoying? Nobody knows

Control Instructions

- One issue: **testing for conditions**
  - Option I: **compare and branch insns**

```
branch-less-than R1,10,target
```

    + Simple, – two ALUs: one for condition, one for target address
  - Option II: **implicit condition codes**

```
subtract R2,R1,10   // sets "negative" CC
branch-neg target
```

    + Condition codes set "for free", – implicit dependence is tricky
  - Option III: **condition registers, separate branch insns**

```
set-less-than R2,R1,10
branch-not-equal-zero R2,target
```

    – Additional instructions, + one ALU per, + explicit dependence

Example: MIPS Conditional Branches

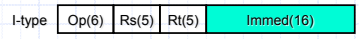
- MIPS uses combination of options I/III
  - Compare 2 registers and branch: **`beq`, `bne`**
    - Equality and inequality only
    + Don't need an adder for comparison
  - Compare 1 register to zero and branch: **`bgtz`, `bgez`, `bltz`, `blez`**
    - Greater/less than comparisons
    + Don't need adder for comparison
  - Set explicit condition registers: **`slt`, `sltu`, `slti`, `sltiu`**, etc.
- Why?
  - More than 80% of branches are (in)equalities or comparisons to 0
  - OK to take two insns to do remaining branches (MCCF)
- Power-PC has separate condition registers and ops

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

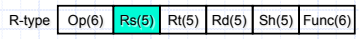
MIPS Control Instructions

- MIPS uses all three
  - PC-relative conditional branches: **`bne`, `beq`, `blez`**, etc.
    - 16-bit relative offset, <0.1% branches need more



  - Absolute jumps unconditional jumps: **`j`**
    - 26-bit offset

  - Indirect jumps: **`jr`**



Control Instructions III

- Another issue: support for procedure calls?
  - Link (remember) address of calling insn + 4 so we can return to it
- MIPS
  - Implicit return address register is $31
  - Direct jump-and-link: **`jal`**
  - Indirect jump-and-link: **`jalr`**

RISC & CISC

- **RISC**: reduced-instruction set computer (coined by Patterson)
- Berkeley RISC-I, Stanford MIPS, & IBM 801
- PowerPC, ARM, SPARC, Alpha, PA-RISC
- Single-cycle execution
- Hardwired control
- Load/store architecture
- Few memory addressing modes
- Fixed instruction format
- Reliance on compiler optimizations

- CISC: complex-instruction set computer (coined by Patterson)
-
- x86, VAX, Motorola 68000, etc.
- Many multicycle operations
- Microcoded multi-cycle operations
- Register-memory & memory-memory
- Many addressing modes
- Many formats and lengths
- Assembly for best performance

Current Winner (units sold): ARM

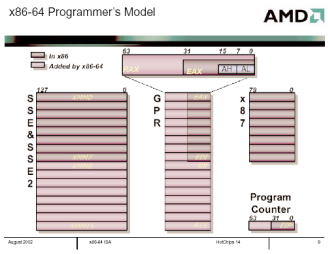
- ARM (Advanced RISC Machine)
  - First ARM chip in mid-1980s (from Acorn Computer Ltd).
  - Over 10 billion units sold (75% of 32/64-bit CPUs)
  - Low-power and embedded devices (iPod, for example)
- 32-bit RISC ISA
  - 16 registers
  - Many addressing modes (for example, auto increment)
  - Condition codes, each instruction can be conditional
- Multiple compatible implementations
  - Intel's X-scale (was DEC's)
  - Others: Freescale (was Motorola), IBM, Texas Instruments, Nintendo, STMicroelectronics, Samsung, Sharp, Philips, etc.
- "Thumb" 16-bit wide instructions
  - Increase code density

Current Winner (revenue): x86

- x86 was first 16-bit chip by ~2 years
  - IBM put it into its PCs because there was no competing choice
  - Rest is historical inertia and "financial feedback"
- *x86 is "Difficult to explain and impossible to love"*
- Complex architecture due to "growth"
  - Typical of many older ISAs, e.g. IBM 360/370/390
  - Started as 16-bit microprocessor (later, 32-bits)
  - Upward compatible from 8080 (accumulator-based)

x86: Registers

- 4 arithmetic,
- 4 address,
- 4 segment,
- 2 control
- Accumulator
  - AH, AL (8 bits)
  - AX (16 bits)
  - EAX (32 bits)
  - RAX (64 bits)

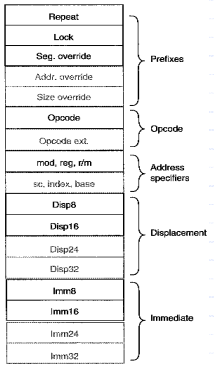


x86 Addressing

- Seven address modes
  - Absolute
  - Register indirect
  - Based
  - Indexed
  - Based indexed with displacement
  - Based with scaled indexed
  - Based with scaled indexed and displacement

x86 Instruction Formats

- Many instruction formats



x86 Outside = RISC Inside

- 1993: Intel wanted out-of-order execution in Pentium Pro
- OOO was very hard to do with a coarse grain ISA like x86
- Their solution? Translate x86 to RISC **uops** in hardware

```
push $eax
```

  is translated (dynamically in hardware) to

```
store $eax [$esp-4]
addi $esp,$esp,-4
```

- Processor maintains **x86 ISA for external compatibility**
- But executes **RISC μISA for internal implementability**
  - Translation itself is proprietary, but 1.6 uops per x86 insn
- Given translator, x86 almost as easy to implement as RISC
- Result: Intel implemented OOO before any RISC company
  - VAX 8800 pioneered μOp conversion w/ 5-stage pipeline in 1987

Transmeta's Take: Code Morphing

- **Code morphing**: x86 translation performed in software
  - Crusoe/Astro are x86 emulators, no actual x86 hardware anywhere
  - Only "code morphing" translation software written in native ISA
  - Native ISA is invisible to applications, OS, even BIOS
  - Different Crusoe versions have (slightly) different ISAs: can't tell
  - How was it done?
    - Code morphing software resides in boot ROM
    - On startup boot ROM hijacks 16MB of main memory
    - Translator loaded into 512KB, rest is **translation cache**
    - Software starts running in **interpreter** mode
    - Interpreter profiles to find "hot" regions: procedures, loops
    - Hot region compiled to native, optimized, cached
    - Gradually, more and more of application starts running native

Emulation/Binary Translation

- Compatibility is still important but definition has changed
  - Less necessary that processor ISA be compatible
  - As long as some combination of ISA + software translation layer is
  - Advances in emulation, binary translation have made this possible
  - **Binary-translation**: transform static image, run native
  - **Emulation**: unmodified image, interpret each dynamic insn
    - Typically optimized with just-in-time (JIT) compilation
  - Examples
    - FX!32: x86 on Alpha
    - IA32EL: x86 on IA64
    - Rosetta: PowerPC on x86
  - Downside: performance overheads

Virtual ISAs

- Java, .Net, and C# use an ISA-like interface
  - Java and .Net use stack-based bytecodes
  - C# has the CLR (common language runtime)
  - Higher-level than machine ISA
    - Design for translation (not direct execution)
  - Goals:
    - Portability (abstract away the actual hardware)
    - Target for high-level compiler (one per language)
    - Source for low-level translator (one per ISA)
    - Flexibility over time
- May allow ISA research to overcome compatibility "gorilla"
  - But Intel wants x86 to be the winning "virtual ISA"

Summary

- What makes a good ISA
  - {Programm|Implement|Compat}-ability
  - Compatibility is a powerful force
  - Compatibility and implementability: μISAs, binary translation
- Aspects of ISAs
- CISC and RISC