CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 2 - Simple Machine Implementations

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152
Last Time in Lecture 1

- Computer Architecture >> ISAs and RTL
  - CS152 is about interaction of hardware and software, and design of appropriate abstraction layers

- Technology and Applications shape Computer Architecture
  - History provides lessons for the future

- First 130 years of CompArch, from Babbage to IBM 360
  - Move from calculators (no conditionals) to fully programmable machines
  - Rapid change started in WWII (mid-1940s), move from electro-mechanical to pure electronic processors

- Cost of software development becomes a large constraint on architecture (need compatibility)

- IBM 360 introduces notion of “family of machines” running same ISA but very different implementations
  - Six different machines released on same day (April 7, 1964)
  - “Future-proofing” for subsequent generations of machine
Instruction Set Architecture (ISA)

- The contract between software and hardware
- Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state
- IBM 360 was first line of machines to separate ISA from implementation (aka. microarchitecture)
- Many implementations possible for a given ISA
  - E.g., the Soviets build code-compatible clones of the IBM360, as did Amdahl after he left IBM.
  - E.g.2., today you can buy AMD or Intel processors that run the x86-64 ISA.
  - E.g.3: many cellphones use the ARM ISA with implementations from many different companies including Apple, Qualcomm, Samsung, Huawei, etc.
- We use Berkeley RISC-V as standard ISA in class
  - www.riscv.org
ISA to Microarchitecture Mapping

- ISA often designed with particular microarchitectural style in mind, e.g.,
  - Accumulator ⇒ hardwired, unpipelined
  - CISC ⇒ microcoded
  - RISC ⇒ hardwired, pipelined
  - VLIW ⇒ fixed-latency in-order parallel pipelines
  - JVM ⇒ software interpretation

- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Spike: Software-interpreted RISC-V machine
  - ARM Jazelle: A hardware JVM processor
  - This lecture: a microcoded RISC-V machine
Why Learn Microprogramming?

- To show how to build very small processors with complex ISAs
- To help you understand where CISC* machines came from
- Because still used in common machines (x86, IBM360, PowerPC)
- As a gentle introduction into machine structures
- To help understand how technology drove the move to RISC*

* “CISC”/”RISC” names much newer than style of machines they refer to.
Control versus Datapath

- Processor designs can be split between *datapath*, where numbers are stored and arithmetic operations computed, and *control*, which sequences operations on datapath.

- Biggest challenge for early computer designers was getting control circuitry correct.
- Maurice Wilkes invented the idea of microprogramming to design the control unit of a processor for EDSAC-II, 1958.
  - Foreshadowed by Babbage’s “Barrel” and mechanisms in earlier programmable calculators.
Microcoded CPU

- **Microcode ROM**: (holds fixed μcode instructions)
- **Datapath**
- **Main Memory**: (holds user program written in macroinstructions, e.g., x86, RISC-V)
Technology Influence

- When microcode appeared in 50s, different technologies for:
  - Logic: Vacuum Tubes
  - Main Memory: Magnetic cores
  - Read-Only Memory: Diode matrix, punched metal cards, ...

- Logic very expensive compared to ROM or RAM
- ROM cheaper than RAM
- ROM much faster than RAM
RISC-V ISA

- New fifth-generation RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- Both 32-bit (RV32) and 64-bit (RV64) address-space variants
- Designed for multiprocessing
- Efficient instruction encoding
- Easy to subset/extend for education/research
- Techreport with RISC-V spec available on class website
- Increasing momentum with industry adoption

Please see CS61C Fall 2017, Lectures 5-7 for RISC-V ISA review:
http://inst.eecs.berkeley.edu/~cs61c/fa17/
RV32 Processor State

Program counter (pc)

32x32-bit integer registers (x0-x31)
  • x0 always contains a 0

32 floating-point (FP) registers (f0-f31)
  • each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

FP status register (fsr), used for FP rounding mode & exception reporting
RISC-V Instruction Encoding

- Can support variable-length instructions.
- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits = $11_2$
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)
### RISC-V Instruction Formats

**R-type**
- **funct7**: Additional opcode bits/immediate
- **rs2**
- **rs1**
- **funct3**
- **rd**
- **opcode**

**I-type**
- **imm[11:0]**
- **rs1**
- **funct3**
- **rd**
- **opcode**

**S-type**
- **imm[11:5]**
- **rs2**
- **rs1**
- **funct3**
- **imm[4:0]**
- **opcode**

**B-type**
- **imm[12]**
- **imm[10:5]**
- **rs2**
- **rs1**
- **funct3**
- **imm[4:1]**
- **imm[11]**
- **opcode**

**U-type**
- **imm[31:12]**
- **rd**
- **opcode**

**J-type**
- **imm[20]**
- **imm[10:1]**
- **imm[11]**
- **imm[19:12]**
- **rd**
- **opcode**

7-bit opcode field (but low 2 bits = 11₂)
Microinstructions written as register transfers:

- MA:=PC means RegSel=PC; RegW=0; RegEn=1; MALd=1
- B:=Reg[rs2] means RegSel=rs2; RegW=0; RegEn=1; BLd=1
- Reg[rd]:=A+B means ALUop=Add; ALUEn=1; RegSel=rd; RegW=1
RISC-V Instruction Execution Phases

- Instruction Fetch
- Instruction Decode
- Register Fetch
- ALU Operations
- *Optional* Memory Operations
- *Optional* Register Writeback
- Calculate Next Instruction Address
Microcode Sketches (1)

Instruction Fetch:
MA,A:=PC
PC:=A+4
\textit{wait for memory}
IR:=\text{Mem}
\textit{dispatch on opcode}

ALU:
A:=\text{Reg}[rs1]
B:=\text{Reg}[rs2]
\text{Reg}[rd]:=\text{ALUOp}(A,B)
goto \text{ instruction fetch}

ALUI:
A:=\text{Reg}[rs1]
B:=\text{ImmI} \hspace{1mm} //\text{Sign-extend 12b immediate}
\text{Reg}[rd]:=\text{ALUOp}(A,B)
goto \text{ instruction fetch}
Microcode Sketches (2)

LW:
A:=Reg[rs1]
B:=ImmI  //Sign-extend 12b immediate
MA:=A+B
wait for memory
Reg[rd]:=Mem
goto instruction fetch

JAL:
Reg[rd]:=A  // Store return address
A:=A-4     // Recover original PC
B:=ImmJ // Jump-style immediate
PC:=A+B
goto instruction fetch

Branch:
A:=Reg[rs1]
B:=Reg[rs2]
if (!ALUOp(A,B)) goto instruction fetch  //Not taken
A:=PC  //Microcode fall through if branch taken
A:=A-4
B:=ImmB// Branch-style immediate
PC:=A+B
goto instruction fetch
Pure ROM Implementation

- How many address bits?
  \[ |\mu\text{address}| = |\mu\text{PC}| + |\text{opcode}| + 1 + 1 \]

- How many data bits?
  \[ |\text{data}| = |\mu\text{PC}| + |\text{control signals}| = |\mu\text{PC}| + 18 \]

- Total ROM size = \(2^{|\mu\text{address}|} \times |\text{data}|\)
### Pure ROM Contents

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Cond?</th>
<th>Busy?</th>
<th>Data</th>
<th>Control Lines</th>
<th>Next μPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>μPC</td>
<td>Opcode</td>
<td>Cond?</td>
<td>Busy?</td>
<td>Data</td>
<td>Control Lines</td>
<td>Next μPC</td>
</tr>
<tr>
<td>fetch0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>MA,A:=PC</td>
<td></td>
<td>fetch1</td>
</tr>
<tr>
<td>fetch1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td></td>
<td></td>
<td>fetch1</td>
</tr>
<tr>
<td>fetch1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>IR:=Mem</td>
<td></td>
<td>fetch2</td>
</tr>
<tr>
<td>fetch2</td>
<td>ALU</td>
<td>X</td>
<td>X</td>
<td>PC:=A+4</td>
<td></td>
<td>ALU0</td>
</tr>
<tr>
<td>fetch2</td>
<td>ALUI</td>
<td>X</td>
<td>X</td>
<td>PC:=A+4</td>
<td></td>
<td>ALUI0</td>
</tr>
<tr>
<td>fetch2</td>
<td>LW</td>
<td>X</td>
<td>X</td>
<td>PC:=A+4</td>
<td></td>
<td>LW0</td>
</tr>
</tbody>
</table>

#### ALU0
- A:=Reg[rs1]  ALU1

#### ALU1
- B:=Reg[rs2]  ALU2

#### ALU2
- Reg[rd]:=ALUOp(A,B)  ALU1
Single-Bus Microcode RISC-V ROM Size

- Instruction fetch sequence 3 common steps
- ~12 instruction groups
- Each group takes ~5 steps (1 for dispatch)
- Total steps $3 + 12 \times 5 = 63$, needs 6 bits for µPC

- Opcode is 5 bits, ~18 control signals

- Total size $= 2^{(6+5+2)} \times (6+18) = 2^{13} \times 24 = \sim 25\text{KiB}$!
Reducing Control Store Size

- Reduce ROM height (#address bits)
  - Use external logic to combine input signals
  - Reduce #states by grouping opcodes

- Reduce ROM width (#data bits)
  - Restrict µPC encoding (next, dispatch, wait on memory,...)
  - Encode control signals (vertical µcoding, nanocoding)
Single-Bus RISC-V Microcode Engine

μPC jump = next | spin | fetch | dispatch | ftrue | ffalse
μPC Jump Types

- *next* increments μPC
- *spin* waits for memory
- *fetch* jumps to start of instruction fetch
- *dispatch* jumps to start of decoded opcode group
- *future/false* jumps to fetch if Cond? true/false
## Encoded ROM Contents

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>µPC</td>
<td>Control Lines</td>
</tr>
<tr>
<td>fetch0</td>
<td>MA,A:=PC</td>
</tr>
<tr>
<td>fetch1</td>
<td>IR:=Mem</td>
</tr>
<tr>
<td>fetch2</td>
<td>PC:=A+4</td>
</tr>
<tr>
<td>ALU0</td>
<td>A:=Reg[rs1]</td>
</tr>
<tr>
<td>ALU1</td>
<td>B:=Reg[rs2]</td>
</tr>
<tr>
<td>ALU2</td>
<td>Reg[rd]:=ALUOp(A,B)</td>
</tr>
<tr>
<td>Branch0</td>
<td>A:=Reg[rs1]</td>
</tr>
<tr>
<td>Branch1</td>
<td>B:=Reg[rs2]</td>
</tr>
<tr>
<td>Branch2</td>
<td>A:=PC</td>
</tr>
<tr>
<td>Branch3</td>
<td>A:=A-4</td>
</tr>
<tr>
<td>Branch4</td>
<td>B:=ImmB</td>
</tr>
<tr>
<td>Branch5</td>
<td>PC:=A+B</td>
</tr>
</tbody>
</table>
CS152 Administrivia

- **Grading clarifications**
  - You must complete 3/5 labs or get an automatic F regardless of other grades

- **Slip days**
  - Problem sets have no slip days
  - Labs have two free extensions (max one per lab) until next class after due date
  - No other extensions without documented emergency
CS252 Administrivia

- CS252 Readings on Website
  - Must use Piazza to send private note on each per paper thread to instructors before midnight Sunday before Monday discussion containing paper report:
    - Write one paragraph on main content of paper including good/bad points of paper
    - Also, 1-3 questions about paper for discussion
    - First two “360 Architecture”, “B5000 Architecture”

- CS252 Project Timeline
  - Proposal due Sunday midnight March 4th
  - One page including:
    - project title
    - team members (2 per project)
    - what problem are you trying to solve?
    - what is your approach?
    - infrastructure to be used
    - timeline/milestones
Implementing Complex Instructions


<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu \text{PC} )</td>
<td>Control Lines</td>
</tr>
<tr>
<td>MMA0</td>
<td>MA:=Reg[rs1]</td>
</tr>
<tr>
<td>MMA1</td>
<td>A:=Mem</td>
</tr>
<tr>
<td>MMA2</td>
<td>MA:=Reg[rs2]</td>
</tr>
<tr>
<td>MMA3</td>
<td>B:=Mem</td>
</tr>
<tr>
<td>MMA4</td>
<td>MA:=Reg[rd]</td>
</tr>
<tr>
<td>MMA5</td>
<td>Mem:=ALUOp(A,B)</td>
</tr>
<tr>
<td>MMA6</td>
<td></td>
</tr>
</tbody>
</table>

Complex instructions usually do not require datapath modifications, only extra space for control program.

Very difficult to implement these instructions using a hardwired controller without substantial datapath modifications.
Single-Bus Datapath for Microcoded RISC-V

Datapath unchanged for complex instructions!
Horizontal vs Vertical μCode

- **Horizontal μcode has wider μinstructions**
  - Multiple parallel operations per μinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits

- **Vertical μcode has narrower μinstructions**
  - Typically a single datapath operation per μinstruction
    - separate μinstruction for branches
  - More microcode steps per macroinstruction
  - More compact ⇒ less bits

- **Nanocoding**
  - Tries to combine best of horizontal and vertical μcode
Nanocoding

Exploits recurring control signal patterns in µcode, e.g.,

ALU0 \( A \leftarrow \text{Reg[rs1]} \)

...  

ALUI0 \( A \leftarrow \text{Reg[rs1]} \)

...  

- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals
## Microprogramming in IBM 360

<table>
<thead>
<tr>
<th></th>
<th>M30</th>
<th>M40</th>
<th>M50</th>
<th>M65</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath width (bits)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>µinst width (bits)</td>
<td>50</td>
<td>52</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>µcode size (K µinsts)</td>
<td>4</td>
<td>4</td>
<td>2.75</td>
<td>2.75</td>
</tr>
<tr>
<td>µstore technology</td>
<td>CCROS</td>
<td>TCROS</td>
<td>BCROS</td>
<td>BCROS</td>
</tr>
<tr>
<td>µstore cycle (ns)</td>
<td>750</td>
<td>625</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>memory cycle (ns)</td>
<td>1500</td>
<td>2500</td>
<td>2000</td>
<td>750</td>
</tr>
<tr>
<td>Rental fee ($K/month)</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>35</td>
</tr>
</tbody>
</table>

- Only the fastest models (75 and 95) were hardwired
IBM Card-Capacitor Read-Only Storage

Punched Card with metal film

Fixed sensing plates

[ IBM Journal, January 1961]
Microcode Emulation

- IBM initially miscalculated the importance of software compatibility with earlier models when introducing the 360 series
- Honeywell stole some IBM 1401 customers by offering translation software (“Liberator”) for Honeywell H200 series machine
- IBM retaliated with optional additional microcode for 360 series that could emulate IBM 1401 ISA, later extended for IBM 7000 series
  - one popular program on 1401 was a 650 simulator, so some customers ran many 650 programs on emulated 1401s
  - (650 simulated on 1401 emulated on 360)
Microprogramming thrived in ‘60s and ‘70s

- Significantly faster ROMs than DRAMs were available
- For complex instruction sets, datapath and controller were cheaper and simpler
- New instructions, e.g., floating point, could be supported without datapath modifications
- Fixing bugs in the controller was easier
- ISA compatibility across various models could be achieved easily and cheaply

Except for the cheapest and fastest machines, all computers were microprogrammed
Microprogramming: early Eighties

- Evolution bred more complex micro-machines
  - Complex instruction sets led to need for subroutine and call stacks in µcode
  - Need for fixing bugs in control programs was in conflict with read-only nature of µROM
  - ➔ Writable Control Store (WCS) (B1700, QMachine, Intel i432, ...)

- With the advent of VLSI technology assumptions about ROM & RAM speed became invalid ➔ more complexity

- Better compilers made complex instructions less important.

- Use of numerous micro-architectural innovations, e.g., pipelining, caches and buffers, made multiple-cycle execution of reg-reg instructions unattractive
VAX 11-780 Microcode
Writable Control Store (WCS)

- Implement control store in RAM not ROM
  - MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 2-10x slower)
  - Bug-free microprograms difficult to write

- User-WCS provided as option on several minicomputers
  - Allowed users to change microcode for each processor

- User-WCS failed
  - Little or no programming tools support
  - Difficult to fit software into small space
  - Microcode control tailored to original ISA, less useful for others
  - Large WCS part of processor state - expensive context switches
  - Protection difficult if user can change microcode
  - Virtual memory required restartable microcode
Analyzing Microcoded Machines

- John Cocke and group at IBM
  - Working on a simple pipelined processor, 801, and advanced compilers inside IBM
  - Ported experimental PL.8 compiler to IBM 370, and only used simple register-register and load/store instructions similar to 801
  - Code ran faster than other existing compilers that used all 370 instructions! (up to 6MIPS whereas 2MIPS considered good before)

- Emer, Clark, at DEC
  - Measured VAX-11/780 using external hardware
  - Found it was actually a 0.5MIPS machine, although usually assumed to be a 1MIPS machine
  - Found 20% of VAX instructions responsible for 60% of microcode, but only account for 0.2% of execution time!

- VAX8800
  - Control Store: 16K*147b RAM, Unified Cache: 64K*8b RAM
  - 4.5x more microstore RAM than cache RAM!
“Iron Law” of Processor Performance

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends on ISA and microarchitecture
- Time per cycle depends upon the microarchitecture and base technology
CPI for Microcoded Machine

- **Inst 1**: 7 cycles
- **Inst 2**: 5 cycles
- **Inst 3**: 10 cycles

Total clock cycles = \(7 + 5 + 10 = 22\)
Total instructions = \(3\)
CPI = \(\frac{22}{3} = 7.33\)

CPI is always an average over a large number of instructions.
IC Technology Changes Tradeoffs

- Logic, RAM, ROM all implemented using MOS transistors
- Semiconductor RAM ~ same speed as ROM
Reconsidering Microcode Machine

Motorola 68000 had 17-bit μcode containing either 10-bit μjump or 9-bit nanoinstruction pointer

- Nanoinstructions were 68 bits wide, decoded to give 196 control signals
From CISC to RISC

- Use fast RAM to build fast instruction *cache* of user-visible instructions, not fixed hardware microroutines
  - Contents of fast instruction memory change to fit what application needs right now

- Use simple ISA to enable hardwired pipelined implementation
  - Most compiled code only used a few of the available CISC instructions
  - Simpler encoding allowed pipelined implementations

- Further benefit with integration
  - In early ‘80s, could finally fit 32-bit datapath + small caches on a single chip
  - No chip crossings in common case allows faster operation
Berkeley RISC Chips

RISC-I (1982) Contains 44,420 transistors, fabbed in 5 µm NMOS, with a die area of 77 mm², ran at 1 MHz. This chip is probably the first VLSI RISC.

RISC-II (1983) contains 40,760 transistors, was fabbed in 3 µm NMOS, ran at 3 MHz, and the size is 60 mm².

Stanford built some too…
Microprogramming is far from extinct

- Played a crucial role in micros of the Eighties
  - DEC uVAX, Motorola 68K series, Intel 286/386

- Plays an assisting role in most modern micros
  - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, ...
  - Most instructions executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke microcode

- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load \( \mu \) code patches at bootup
  - Intel had to scramble to resurrect microcode tools and find original microcode engineers to patch Meltdown/Spectre security vulnerabilities
Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252