Last Time in Lecture 2

- Stack machines popular to simplify High-Level Language (HLL) implementation
  - Algol-68 & Burroughs B5000, Forth machines, Occam & Transputers, Java VMs & Java Interpreters
- General-purpose register machines provide greater efficiency with better compiler technology (or assembly coding)
  - Compilers can explicitly manage fastest level of memory hierarchy (registers)
- Microcoding was a straightforward way to implement simple machines with low gate count
  - But also allowed arbitrary instruction complexity as microcode stores grew
  - Makes most sense when fast read-only memory (ROM) significantly faster than read-write memory (RAM)
Microprogramming thrived in the Seventies

- Significantly faster ROMs than DRAMs/core were available
- For complex instruction sets (CISC), 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*

Writable Control Store (WCS)

- Implement control store in RAM not ROM
  - MOS SRAM memories now became 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
Microprogramming: early Eighties

- Evolution bred more complex micro-machines
  - CISC ISAs 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
  - --> 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

Microprogramming in Modern Usage

- *Microprogramming is far from extinct*
- Played a crucial role in micros of the Eighties
  
  *DEC uVAX, Motorola 68K series, Intel 386 and 486*

- Microcode pays an assisting role in most modern micros (*AMD Athlon, Intel Core 2 Duo, IBM PowerPC*)
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine

- *Patchable* microcode common for post-fabrication bug fixes, e.g. Intel Pentiums load μcode patches at bootup
From CISC to RISC

- Use fast RAM to build fast instruction cache of user-visible instructions, not fixed hardware microroutines
  - Can change contents of fast instruction memory 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, can fit 32-bit datapath + small caches on a single chip
  - No chip crossings in common case allows faster operation

Horizontal vs Vertical µCode

- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction for branches
  - More steps to 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.,

\[ \text{ALU}_0 \ A \leftarrow \text{Reg}[rs] \]
\[ ... \]
\[ \text{ALU}_i \ A \leftarrow \text{Reg}[rs] \]

- MC68000 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

CDC 6600 *Seymour Cray, 1964*

- A fast pipelined machine with 60-bit words
- Ten functional units
  - Floating Point: adder, multiplier, divider
  - Integer: adder, multiplier
  ...
- Hardwired control (no microcoding)
- Dynamic scheduling of instructions using a scoreboard
- Ten Peripheral Processors for Input/Output
  - a fast time-shared 12-bit integer ALU
- Very fast clock, 10MHz
- Novel freon-based technology for cooling
CDC 6600: Datapath

Central Memory
128K words, 32 banks, 1µs cycle

Address Regs 8 x 18-bit

Index Regs 8 x 18-bit

Operand Regs 8 x 60-bit

10 Functional Units

Inst. Stack 8 x 60-bit

CDC 6600: A Load/Store Architecture

- Separate instructions to manipulate three types of reg.
  8 60-bit data registers (X)
  8 18-bit address registers (A)
  8 18-bit index registers (B)

- All arithmetic and logic instructions are reg-to-reg

  \[
  \text{opcode} | i | j | k \quad \text{Ri} \leftarrow (Rj) \text{ op } (Rk)
  \]

- Only Load and Store instructions refer to memory!

  \[
  \text{opcode} | i | j | \text{disp} \quad \text{Ri} \leftarrow M[(Rj) + \text{disp}]
  \]

  Touching address registers 1 to 5 initiates a load
  6 to 7 initiates a store
  - very useful for vector operations
CDC6600: Vector Addition

B0 ← - n

loop: JZE B0, exit
A0 ← B0 + a0        load X0
A1 ← B0 + b0        load X1
X6 ← X0 + X1
A6 ← B0 + c0        store X6
B0 ← B0 + 1

jump loop

Ai = address register
Bi = index register
Xi = data register

CDC6600 ISA designed to simplify high-performance implementation

- Use of three-address, register-register ALU instructions simplifies pipelined implementation
  - No implicit dependencies between inputs and outputs
- Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses
  - Software can schedule load of address register before use of value
  - Can interleave independent instructions inbetween
- CDC6600 has multiple parallel but unpipelined functional units
  - E.g., 2 separate multipl
- Follow-on machine CDC7600 used pipelined functional units
  - Foreshadows later RISC designs
### Instruction Set Architecture (ISA) versus Implementation

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines instruction format (bit encoding) and instruction semantics
  - Examples: MIPS, x86, IBM 360, JVM

- Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), z990 (c. 2004)
  - x86 implementations: 8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4 (c. 2000), AMD Athlon, Transmeta Crusoe, SoftPC
  - MIPS implementations: R2000, R4000, R10000, ...
  - JVM: HotSpot, PicoJava, ARM Jazelle, ...

### "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 upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>cycle time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcoded</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Single-cycle unpipelined</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>Pipelined</td>
<td>1</td>
<td>short</td>
</tr>
</tbody>
</table>

this lecture
Hardware Elements

- Combinational circuits
  - Mux, Decoder, ALU, ...

- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM

Edge-triggered: Data is sampled at the rising edge

Register Files

- Reads are combinational
Register File Implementation

- Register files with a large number of ports are difficult to design
  - Almost all MIPS instructions have exactly 2 register source operands
  - Intel’s Itanium, GPR File has 128 registers with 8 read ports and 4 write ports!!!
CS152 Administrivia

• Krste, no office hours this Monday (ISSCC) - email for alternate time
• Henry office hours, location?
  – 9:30-10:30AM Mondays
  – 2:00-3:00PM Fridays
• First lab and problem sets coming out soon (by Tuesday’s class)

Implementing MIPS:
Single-cycle per instruction datapath & control logic
The MIPS ISA

Processor State
- 32 32-bit GPRs, R0 always contains a 0
- 32 single precision FPRs, may also be viewed as 16 double precision FPRs
- FP status register, used for FP compares & exceptions
- PC, the program counter
- some other special registers

Data types
- 8-bit byte, 16-bit half word
- 32-bit word for integers
- 32-bit word for single precision floating point
- 64-bit word for double precision floating point

Load/Store style instruction set
- data addressing modes- immediate & indexed
- branch addressing modes- PC relative & register indirect
- Byte addressable memory- big endian mode

All instructions are 32 bits

Instruction Execution

Execution of an instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back

and the computation of the address of the next instruction
Datapath: Reg-Reg ALU Instructions

RegWrite Timing?

rd ← (rs) func (rt)

Datapath: Reg-Imm ALU Instructions

rt ← (rs) op immediate
Conflicts in Merging Datapath

Datapath for ALU Instructions
**Datapath for Memory Instructions**

Should program and data memory be separate?

*Harvard style: separate* (Aiken and Mark 1 influence)
- read-only program memory
- read/write data memory

- Note:
  Somehow there must be a way to load the program memory

*Princeton style: the same* (von Neumann’s influence)
- single read/write memory for program and data

- Note:
  A Load or Store instruction requires accessing the memory more than once during its execution

---

**Load/Store Instructions: Harvard Datapath**

![Diagram of Load/Store Instructions](image)

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25</td>
<td>21 20 16 15 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **opcode**
  - 6
  - rs
  - rt
- **displacement**
  - 16
- **addressing mode**
  - (rs) + displacement

rs is the base register
rt is the destination of a Load or the source for a Store
### MIPS Control Instructions

#### Conditional (on GPR) PC-relative branch

6 5 5 16

- **opcode**
- **rs**
- **offset**

BEQZ, BNEZ

#### Unconditional register-indirect jumps

6 5 5 16

- **opcode**
- **rs**

JR, JALR

#### Unconditional absolute jumps

6 26

- **opcode**
- **target**

J, JAL

- PC-relative branches add offset×4 to PC+4 to calculate the target address (offset is in words): ±128 KB range
- Absolute jumps append target×4 to PC<31:28> to calculate the target address: 256 MB range
- Jump-&-link stores PC+4 into the link register (R31)
- All Control Transfers are delayed by 1 instruction

*we will worry about the branch delay slot later*

---

### Conditional Branches (BEQZ, BNEZ)

![Conditional Branches Diagram](image-url)
Register-Indirect Jumps (JR)

Register-Indirect Jump-&-Link (JALR)
Absolute Jumps (J, JAL)

Harvard-Style Datapath for MIPS
Hardwired Control is pure Combinational Logic

ALU Control & Immediate Extension
## Hardwired Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ExtSel</th>
<th>BSrc</th>
<th>OpSel</th>
<th>MemW</th>
<th>RegW</th>
<th>WBSrc</th>
<th>RegDst</th>
<th>PCSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>*</td>
<td>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUi</td>
<td>sExt16</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUiu</td>
<td>uExt16</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>LW</td>
<td>sExt16</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>SW</td>
<td>sExt16</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQZ&lt;z&gt;</td>
<td>sExt16</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQZ&lt;1&gt;</td>
<td>sExt16</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>J</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>jabs</td>
</tr>
<tr>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>jabs</td>
</tr>
<tr>
<td>JR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>rind</td>
</tr>
<tr>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>rind</td>
</tr>
</tbody>
</table>

BSrc = Reg / Imm  WBSrc = ALU / Mem / PC  
RegDst = rt / rd / R31  PCSrc = pc+4 / br / rind / jabs

## Single-Cycle Hardwired Control: Harvard architecture

We will assume

- clock period is sufficiently long for all of the following steps to be “completed”:

  1. instruction fetch  
  2. decode and register fetch  
  3. ALU operation  
  4. data fetch if required  
  5. register write-back setup time

\[ t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB} \]

- At the rising edge of the following clock, the PC, the register file and the memory are updated
An Ideal Pipeline

- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

*These conditions generally hold for industrial assembly lines.*
*But can an instruction pipeline satisfy the last condition?*

---

Pipelined MIPS

To pipeline MIPS:

- First build MIPS without pipelining with CPI=1
- Next, add pipeline registers to reduce cycle time while maintaining CPI=1
Pipelined Datapath

Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max \{ t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW} \} = t_{DM} \text{ probably} \]

However, CPI will increase unless instructions are pipelined

How to divide the datapath into stages

Suppose memory is significantly slower than other stages. In particular, suppose

\[ t_{IM} = 10 \text{ units} \]
\[ t_{DM} = 10 \text{ units} \]
\[ t_{ALU} = 5 \text{ units} \]
\[ t_{RF} = 1 \text{ unit} \]
\[ t_{RW} = 1 \text{ unit} \]

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance
Alternative Pipelining

\[ t_C > \max\{t_{IM}, t_{RF} + t_{ALU}, t_{DM} + t_{RW}\} = t_{DM} + t_{RW} \]

\[ \Rightarrow \text{increase the critical path by 10\%} \]

Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase

Maximum Speedup by Pipelining

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Unpipelined ( t_C )</th>
<th>Pipelined Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( t_{IM} = t_{DM} = 10, t_{ALU} = 5, t_{RF} = t_{RW} = 1 ) 4-stage pipeline</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>2. ( t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5 ) 4-stage pipeline</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>3. ( t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5 ) 5-stage pipeline</td>
<td>25</td>
<td>5</td>
</tr>
</tbody>
</table>

It is possible to achieve higher speedup with more stages in the pipeline.
Summary

- Microcoding became less attractive as gap between RAM and ROM speeds reduced
- Complex instruction sets difficult to pipeline, so difficult to increase performance as gate count grew
- Iron-law explains architecture design space
  - Trade instruction/program, cycles/instruction, and time/cycle
- Load-Store RISC ISAs designed for efficient pipelined implementations
  - Very similar to vertical microcode, inspired by earlier Cray machines
- MIPS ISA will be used in class and problems, SPARC in lab (two very similar ISAs)

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