CPU architecture
Topics

• CPU stages and design
• Pipelining
• Y86 semantics
A CPU consists of
- Combinational circuits for computation
- Sequential circuits for (cached) memory
- Wires/buses for connectivity and intermediate results
- A clocked register PC for synchronization
Example

<table>
<thead>
<tr>
<th>Cycle 1:</th>
<th>0x000:  irmovq $0x100,%rbx  # %rbx &lt;= 0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 2:</td>
<td>0x00a:  irmovq $0x200,%rdx  # %rdx &lt;= 0x200</td>
</tr>
<tr>
<td>Cycle 3:</td>
<td>0x014:  addq %rdx,%rbx  # %rbx &lt;= 0x300 CC &lt;= 000</td>
</tr>
<tr>
<td>Cycle 4:</td>
<td>0x016:  je dest  # Not taken</td>
</tr>
<tr>
<td>Cycle 5:</td>
<td>0x01f:  rmmovq %rbx,0(%rdx)  # M[0x200] &lt;= 0x300</td>
</tr>
</tbody>
</table>

① Beginning of cycle 3

② End of cycle 3
Example

Clock

| Cycle 1: | 0x000: irmovq $0x100,%rbx # %rbx <-- 0x100 |
| Cycle 2: | 0x00a: irmovq $0x200,%rdx # %rdx <-- 0x200 |
| Cycle 3: | 0x014: addq %rdx,%rbx # %rbx <-- 0x300 CC <-- 000 |
| Cycle 4: | 0x016: je dest # Not taken |
| Cycle 5: | 0x01f: rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300 |

③ Beginning of cycle 4

④ End of cycle 4
CPU design

- **SEQ**: sequential Y86 CPU
  - Runs one instruction at a time
  - *ysim*: simulator

- **Components**:
  - Clocked register (PC)
  - Hardware units (blue boxes)
    - Combinational/sequential circuits
    - ALU, register file, memory
  - Control logic (grey rectangles)
    - Combinational circuits
    - Details in textbook
  - Wires (white circles)
    - Word (thick lines)
    - Byte (thin lines)
    - Bit (dotted lines)

- **Principle**: no reading back
  - von Neumann stages run simultaneously
  - Effects remain internally consistent
CPU measurement

- **Throughput**: instructions executed per second
  - GIPS: billions of (“giga-”) instructions per second
  - 1 GIPS → each instruction takes 1 nanosecond (a billionth of a second)

- **Latency / delay**: time required per instruction
  - Picosecond: $10^{-12}$ seconds    Nanosecond: $10^{-9}$ seconds
  - $1,000 \text{ ps} = 1 \text{ nanosecond}$

- Relationship: $\text{throughput} = \# \text{ instructions} / \text{ latency}$
  - Example: $1 / 320\text{ps} \times (1000\text{ps/ns}) = 0.003125 \times 1000 \approx 3.1 \text{ GIPS}$
System design

- Current CPU design is serial
  - One instruction executes at a time
  - Only way to improve is to run faster!
  - Limited by speed of light / electricity
- One approach: make it smaller
  - Shorter circuit = faster circuit
  - Limited by manufacturing technology

What else could we do?
System design

- Idea: pipelined design
  - Multiple instructions execute simultaneously ("instruction-level parallelism")
  - Similar to cafeteria line or car wash
  - Split logic into stages and connect stages with clocked registers
  - System design tradeoff: throughput vs. latency
It's complicated!
- Split up the stages and add more clocked registers for intermediate results
Pipelining

- Limitation: non-uniform partitioning
  - Logic segments may have significantly different lengths
Pipelining

• Limitation: **dependencies**
  - The effect of one instruction depends on the result of another
  - Both **data** and **control** dependencies
  - Sometimes referred to as **hazards**

**Data dependency:**
- `irmovq $8, %rax`
- `addq %rax, %rbx`
- `mrmovq 0x300(%rbx), %rdx`

**Control dependency:**
- `loop:
  - subq %rdx, %rbx
  - jne loop
  - irmovq $10, %rdx`
Pipelining

- Approaches to avoiding hazards
  - **Stalling**: “hold back” an instruction temporarily
  - **Data forwarding**: allow latter stages to feed into earlier stages, bypassing memory or registers
  - Hybrid: stall and forward
  - **Branch prediction**: guess address of next instruction
  - **Halt** execution (or throw an exception)
- For more info, read CS:APP section 4.5
Conditional moves

• Similar to conditional jumps, but they move data if certain condition codes are set
  – Benefit: no branch prediction penalty
    • Improved performance in the presence of pipelining

\[
\text{if } (a > b) \ c = d; \\
\]

```
subq %rbx, %rax
jle skip
rrmovq %rdx, %rcx
skip:
subq %rbx, %rax
cmovg %rdx, %rcx
```

Data (CCs) and control dependencies

No control dependency (only data)
Summary

• We’ve now learned how a CPU is constructed
  – Transistors → logic gates → circuits → CPU
  – Pipelining provides instruction-level parallelism
• This is not a CPU architecture class
  – We won’t be closely studying the specifics of SEQ
  – If you’re interested, the details are in section 4.3
  – Same for PIPE (the pipelined version), in section 4.5
  – If you’re REALLY interested, plan to take CS 456
Course objectives:

- Summarize the construction of a pipelined processor from low-level building blocks
- Describe and categorize hardware techniques for parallel implementation at the instruction, data, and thread levels
- Summarize storage and I/O interfacing techniques
- Apply address decoding and memory hierarchy strategies
- Evaluate the performance impact of various hardware designs, including caches
- Describe how hardware implementations can improve overall system performance
- Justify the use of hardware-based optimizations that fail occasionally
- Compare and contrast the actual execution of code with software designs
- Analyze how a person’s logical flow of thinking (sequential) differs from the processor implementation
- Demonstrate the ability to communicate hardware and software design trade-offs to both professional colleagues and laypeople
Lessons learned

• **Computers are not human**; they’re complex machines
  – Machines require extremely precise inputs
  – Machine output can be difficult to interpret

• **Abstraction helps to manage complexity**
  – Use simpler components to build more complex ones

• **System design involves tradeoffs**
  – Simpler ISA vs. ease of coding
  – Throughput vs. latency

• **The details matter (A LOT!)**
  – There are many ways to fail
  – Skill and dedication are required to succeed
Y86 semantics

- **Semantics**: the study of *meaning*
  - What does an instruction "mean"?
  - For us, it is *the effect that it has on the machine*
  - This requires understanding the purpose and actions of all stages of the Y86-64 CPU
6-stage von Neumann cycle

1) Fetch
2) Decode
3) Execute
4) Memory
5) Write back
6) PC update
• Read ten bytes from memory at address PC
• Extract instruction fields
  – icode and ifun
  – rA and rB
  – valC
• Compute valP (address of next instruction)
  – PC + 1 + needsRegIDs + 8*needsValC
Decode

- Read register file
  - Read srcA into valA
  - Read srcB into valB
Execute

- Perform arithmetic or logic operation
  - Could also be an effective address calculation or stack pointer increment / decrement
  - First input is valC (immediate/offset) or valA (register)
  - Second input is valB (register)
- Set condition codes
  - Only if OPq
Memory

- Read or write memory
  - No instruction does both!
  - Effective address is valE or valA (depending on icode)
  - Data to be written is either valA or valP (depending on icode)
  - Data is read into valM
Write back

- Write register file
  - Write valE (from ALU execute) to dstE for some icodes
  - Write valM (from memory) to dstM for some icodes
  - Use value 0xF to disable one or both write(s) for some icodes
PC update

- Set new PC
  - valP (next instruction) for most icodes
  - Either valP or valC for conditional jumps depending on Cnd
  - valM (return address popped from stack) for ret
Y86 semantics

- **Semantics**: the study of *meaning*
  - For us, it is *the effect that it has on the machine*
  - We should specify these semantics very **formally**
  - This will help us think correctly about P4
  - ISA reference sheet includes mathematical semantics

<table>
<thead>
<tr>
<th>Stage</th>
<th>HALT</th>
<th>NOP</th>
<th>CMOV</th>
<th>IRMOVQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fch</td>
<td><code>icode ← M_1[PC]</code></td>
<td><code>icode ← M_1[PC]</code></td>
<td><code>icode:ifun ← M_1[PC]</code></td>
<td><code>icode:ifun ← M_1[PC]</code></td>
</tr>
<tr>
<td></td>
<td><code>valP ← PC + 1</code></td>
<td><code>valP ← PC + 1</code></td>
<td><code>valP ← PC + 2</code></td>
<td><code>valP ← PC + 10</code></td>
</tr>
<tr>
<td>Dec</td>
<td><code>cpu.stat = HLT</code></td>
<td></td>
<td><code>valA ← R[rA]</code></td>
<td></td>
</tr>
<tr>
<td>Exe</td>
<td></td>
<td></td>
<td><code>valE ← valA</code></td>
<td><code>valE ← valC</code></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td><code>Cnd ← Cond(CC,ifun)</code></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td><code>PC ← 0</code></td>
<td></td>
<td></td>
<td><code>R[rB] ← valE</code></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td><code>PC ← valP</code></td>
<td><code>PC ← valP</code></td>
<td><code>PC ← valP</code></td>
</tr>
</tbody>
</table>
Aside: syntax notes

- \( R[RSP] = \text{the value of} \%rsp \)
- \( R[rA] = \text{the value of register with id} \ rA \)
- \( M_1[PC] = \text{the value of one byte in memory at address} \ PC \)
- \( M_8[PC+2] = \text{the value of eight bytes in memory at address} \ PC+2 \)
- \( rA:rB = M_1[PC+1] \text{ means read the byte at address} \ PC+1 \)
  - Split it into high- and low-order 4-bits for \( rA \) and \( rB \)
- \( \text{Cond}(CC, \text{ifun}) \) returns 0 or 1 based on \( CC \) and \( \text{ifun} \)
  - Determines whether the given CMOV/JUMP should happen
- Convention: write addresses using hex padded to three chars
- Convention: write integer literals using decimal w/ no padding
**Example: IRMOVQ**

<table>
<thead>
<tr>
<th>Stage</th>
<th>IRMOVQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fch</td>
<td>icode:ifun ← M₁[PC]</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M₁[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valC ← M₈[PC+2]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC + 10</td>
</tr>
<tr>
<td>Dec</td>
<td></td>
</tr>
<tr>
<td>Exe</td>
<td>valE ← valC</td>
</tr>
<tr>
<td>Mem</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>R[rB] ← valE</td>
</tr>
<tr>
<td>PC</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>

| icode:ifun | M₁[0x016] = 3:0  |
| rA:rB      | M₁[0x017] = f:4  |
| valC       | M₈[0x018] = 128  |
| valP       | 0x016 + 10 = 0x020 |
| valE       | 128               |
| R[%rsp]    | valE = 128        |
| PC         | valP = 0x020      |

This instruction sets `%rsp` to 128 and increments the PC by 10
Example: POPQ

```
0x02c: b00f

<table>
<thead>
<tr>
<th>popq</th>
<th>%rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[%rsp] = 120</td>
<td>M_8[120] = 9</td>
</tr>
</tbody>
</table>
```

```
Table:

<table>
<thead>
<tr>
<th>Stage</th>
<th>POPQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fch</td>
<td>icode:ifun ← M_1[PC]</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M_1[PC+1]</td>
</tr>
<tr>
<td>Dec</td>
<td>valP ← PC + 2</td>
</tr>
<tr>
<td></td>
<td>valA ← R[RSP]</td>
</tr>
<tr>
<td></td>
<td>valB ← R[RSP]</td>
</tr>
<tr>
<td>Exe</td>
<td>valE ← valB + 8</td>
</tr>
<tr>
<td>Mem</td>
<td>valM ← M_8[valA]</td>
</tr>
<tr>
<td>WB</td>
<td>R[RSP] ← valE</td>
</tr>
<tr>
<td></td>
<td>R[rA] ← valM</td>
</tr>
<tr>
<td>PC</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>
```

```
icode:ifun ← M_1[0x02c] = b:0
rA:rB ← M_1[0x02d] = 0:f
valP ← 0x02c + 2 = 0x02e
valA ← R[%rsp] = 120
valB ← R[%rsp] = 120
valE ← 120 + 8 = 128
valM ← M_8[120] = 9
R[%rsp] ← 128
R[%rax] ← 9
PC ← 0x02e
```

This instruction sets %rax to 9, sets %rsp to 128, and increments the PC by 2
Example: CALL

This instruction sets %rsp to 120, stores the return address 0x040 at [%rsp], and sets the PC to 0x041
### Y86 semantics

<table>
<thead>
<tr>
<th>Stage</th>
<th>HALT</th>
<th>NOP</th>
<th>CMOV</th>
<th>IRMOVQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec</td>
<td>valP ← PC + 1</td>
<td>valP ← PC + 1</td>
<td>valP ← PC + 2</td>
<td>valP ← PC + 10</td>
</tr>
<tr>
<td>Exe</td>
<td>cpu.stat = HLT</td>
<td>valA ← R[rA]</td>
<td>valE ← valA</td>
<td>valE ← valC</td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td>Cnd ← Cond(CC,ifun)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>PC ← 0</td>
<td>PC ← valP</td>
<td>Cnd ? R[rB] ← valE</td>
<td>R[rB] ← valE</td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
<td>R[rB] ← valE</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage</th>
<th>RMOVQ</th>
<th>MMOVQ</th>
<th>OPq</th>
<th>JXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>valP ← PC + 10</td>
<td>valP ← PC + 2</td>
<td>valP ← PC + 9</td>
<td>valP ← PC + 9</td>
</tr>
<tr>
<td>WB</td>
<td>valA ← R[rA]</td>
<td>valB ← R[rB]</td>
<td>valE ← valB + valC</td>
<td>valE ← valB + valC</td>
</tr>
<tr>
<td>PC</td>
<td>valB ← R[rB]</td>
<td>valC ← valB + valC</td>
<td>valE ← valB + valC</td>
<td>valE ← valB + valC</td>
</tr>
<tr>
<td>Cnd ← Cond(CC,ifun)</td>
<td>Set CC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage</th>
<th>CALL</th>
<th>RET</th>
<th>PUSHQ</th>
<th>POPQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec</td>
<td>valC ← M₈[PC+1]</td>
<td>valP ← PC + 1</td>
<td>valP ← PC + 2</td>
<td>valP ← PC + 2</td>
</tr>
<tr>
<td>WB</td>
<td>valE ← valB - 8</td>
<td>valE ← valB - 8</td>
<td>valE ← valB + 8</td>
<td>valE ← valB + 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R[rA] ← valE</td>
<td>R[rA] ← valE</td>
<td>R[rA] ← valE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ← valP</td>
<td>PC ← valM</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>
Y86 CPU (P4)

1) Fetch ← P3!
   ▪ Splits instruction at PC into pieces
   ▪ Save info in y86_inst_t struct

2) Decode (register file)
   ▪ Reads registers
   ▪ P4: Sets valA

3) Execute (ALU)
   ▪ Arithmetic/logic operation, effective address calculation, or stack pointer increment/decrement
   ▪ P4: Sets valE and Cnd

4) Memory (RAM)
   ▪ Reads/writes memory

5) Write back (register file)
   ▪ Sets registers

6) PC update
   ▪ Sets new PC