CS429: Computer Organization and Architecture
Datapath II

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: July 5, 2018 at 11:55
## The ISA

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>halt</strong></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>nop</strong></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA,rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V,rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA,D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB),rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA,rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**SEQ Stages**

**Fetch**: Read instruction from instruction memory.

**Decode**: Read program registers

**Execute**: Compute value or address

**Memory**: Read or write back data.

**Write Back**: Write program registers.

**PC**: Update the program counter.
Predefined Blocks

- **PC**: Register containing the PC.
- **Instruction memory**: Read 10 bytes (PC to PC+9).
- **Split**: Divide instruction byte into icode and ifun.
- **Align**: Get fields for rA, rB, and valC.
Control Logic

- **Instr. Valid:** Is this instruction valid?
- **Needs regids:** Does this instruction have a register byte?
- **Need valC:** Does this instruction have a constant word?
We can define how the various signals are computed using our HCL language:

```c
bool instr_valid = icode in
    { INOP, IHALT, IRRMOVQ, IIRMMOVQ, IRMMOVQ, IMRMOVQ,
      IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };

bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
              IIRMMOVQ, IRMMOVQ, IMRMOVQ };
```
Register File
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 0xF (no access)

Control Logic
- srA, srB: read port addresses
- dstA, dstB: write port addresses

Note that wires in the implementation have different semantics depending on the operation.
Where is register $A$ coming from?

- **OPq rA,rB**
  
  **Decode**
  
  \[
  \text{valA} \leftarrow R[rA]
  \]
  
  **Read operand A**

- **rmmovq rA,D(rB)**
  
  **Decode**
  
  \[
  \text{valA} \leftarrow R[rA]
  \]
  
  **Read operand A**

- **popq rA**
  
  **Decode**
  
  \[
  \text{valA} \leftarrow R[%rsp]
  \]
  
  **Read stack pointer**

- **jXX Dest**
  
  **Decode**
  
  **No operand**

- **call Dest**
  
  **Decode**
  
  **No operand**

- **ret**
  
  **Decode**
  
  \[
  \text{valA} \leftarrow R[%rsp]
  \]
  
  **Read stack pointer**

```c
int srcA = [
   icode in \{ \text{IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ} \}: rA;
   icode in \{ \text{IPOPQ, IRET} \}: \text{RESP};
   1: \text{RNONE} \quad \# \text{Don’t need register}
];
```
Execute Logic

Units

- ALU: Implements the 4 required functions, and generates condition code values.
- CC: Register with 3 condition code bits.
- bcond: computes branch flag.

Control Logic

- Set CC: should condition code register be loaded?
- ALU A: Input A to ALU
- ALU B: Input B to ALU
- ALU fun: What function should ALU compute?
What is feeding the A input to the ALU?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPq rA,rB</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td>valE ← valB OP valA</td>
<td>Perform ALU operation</td>
</tr>
<tr>
<td>rmmovq rA,D(rB)</td>
<td>Compute effective address</td>
</tr>
<tr>
<td>valE ← valB + valC</td>
<td>Compute effective address</td>
</tr>
<tr>
<td>popq rA</td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>valE ← valB + 8</td>
<td>Increment stack pointer</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>No operation</td>
</tr>
<tr>
<td>call Dest</td>
<td>Decrement stack pointer</td>
</tr>
<tr>
<td>valE ← valB + -8</td>
<td>Decrement stack pointer</td>
</tr>
<tr>
<td>ret</td>
<td>Increment stack pointer</td>
</tr>
</tbody>
</table>

```c
int aluA = [
    icode in { IRRMOVQ, IOPQ }: valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ }: valC;
    icode in { ICALL, IPUSHQ }: -8;
    icode in { IRET, IPOPQ }: 8;
    # Other instructions don’t need an ALU
];
```
### ALU Operation

#### What function should the ALU perform?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPq rA,rB</td>
<td>Perform ALU operation (op)</td>
</tr>
<tr>
<td>rmmovq rA,D(rB)</td>
<td>Compute effective address (add)</td>
</tr>
<tr>
<td>popq rA</td>
<td>Increment stack pointer (add)</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>No operation</td>
</tr>
<tr>
<td>call Dest</td>
<td>Decrement stack pointer (add)</td>
</tr>
<tr>
<td>ret</td>
<td>Increment stack pointer (add)</td>
</tr>
</tbody>
</table>

```c
int alufun = [
    icode == IOPQ: ifun;
    1: ALUADD;
];
```
Memory Logic

Memory
- Reads or writes memory word.

Control Logic
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: select address
- Mem. data: select data
What memory address is stored / loaded?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPq rA,rB</td>
<td>No operation</td>
</tr>
<tr>
<td>rmmovq rA,D(rB)</td>
<td>Write value to memory</td>
</tr>
<tr>
<td>popq rA</td>
<td>Read from stack</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>No operation</td>
</tr>
<tr>
<td>call Dest</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td>ret</td>
<td>Increment stack pointer</td>
</tr>
</tbody>
</table>

```c
int mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ }: valE;
    icode in { IPOPQ, IRET }: valA;
    # Other instructions don’t need address
    ];
```
### For what instructions is memory read?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OPq rA,rB</code></td>
<td>Memory</td>
<td>No operation</td>
</tr>
<tr>
<td><code>rmmovq rA,D(rB)</code></td>
<td>Memory</td>
<td>Write value to memory</td>
</tr>
<tr>
<td><code>popq rA</code></td>
<td>Memory</td>
<td>Read from stack</td>
</tr>
<tr>
<td><code>jXX Dest</code></td>
<td>Memory</td>
<td>No operation</td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td>Memory</td>
<td>Write return value on stack</td>
</tr>
<tr>
<td><code>ret</code></td>
<td>Memory</td>
<td>Increment stack pointer</td>
</tr>
</tbody>
</table>

```cpp
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
```
Notice for Write-back, there is no explicit hardware here.

That’s because the location for writing back was determined at the decode stage. At this stage we have simply computed the values to write-back into the register file!
Where to store the value computed by the ALU?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPq rA,rB</td>
<td></td>
<td>Write back result</td>
</tr>
<tr>
<td>Write-back</td>
<td>R[rB] ← valE</td>
<td></td>
</tr>
<tr>
<td>rmmovq rA,D(rB)</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Write-back</td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>popq rA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td>R[%rsp] ← valE</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>jXX Dest</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Write-back</td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>call Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td>R[%rsp] ← valE</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>Write-back</td>
<td>R[%rsp] ← valE</td>
<td>Update stack pointer</td>
</tr>
</tbody>
</table>

```
int dstE = [
    icode in { IRRMOVQ, IIRMMOVQ, IOPQ }: rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET }: RESP;
    1: RNONE  # Don’t need register
];
```
New PC
Select next value of PC.

Depends on:
- \texttt{icode}: current instruction
- \texttt{Bch}: result of branch logic
- \texttt{valC}: constant from instruction word
- \texttt{valM}: value from memory (stack)
- \texttt{valP}: predicted value from fetch
PC Update

What is the new value of the PC?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC Update</th>
<th>PC New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPq rA,rB</td>
<td>PC ← valP</td>
<td>Update PC (by 2)</td>
</tr>
<tr>
<td>rmmovq rA,D(rB)</td>
<td>PC ← valP</td>
<td>Update PC (by 10)</td>
</tr>
<tr>
<td>popq rA</td>
<td>PC ← valP</td>
<td>Update PC (by 2)</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>PC ← Bch ? valC : valP</td>
<td>Update PC (to what?)</td>
</tr>
<tr>
<td>call Dest</td>
<td>PC ← valC</td>
<td>Set PC to destination</td>
</tr>
<tr>
<td>ret</td>
<td>PC ← valM</td>
<td>Set PC to return address</td>
</tr>
</tbody>
</table>

```c
int new_pc = [
    icode == ICALL: valC;
    icode == IJXX && Bch: ValC;
    icode == IRET: valM;
    1: valP;
];
```
State: All updated as the clock rises.
- PC register
- Condition Code register
- Register file

Combinational Logic
- ALU
- Control logic

Sequential Logic
- Instruction memory
- Register file
- Data memory
SEQ Operation 2

Cycle 1: 0x000: irmovq $0x100,%rbx # %rbx <-- 0x100
Cycle 2: 0x006: irmovq $0x200,%rdx # %rdx <-- 0x200
Cycle 3: 0x00c: addq %rdx,%rbx # %rbx <-- 0x300, CC <-- 000
Cycle 4: 0x00e: je dest # Not taken

- state is set according to first irmovq instruction
- combinational logic is starting to react to state changes
<table>
<thead>
<tr>
<th>Cycle 1:</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000:</td>
<td><code>irmovq $0x100,%rbx</code> # %rbx &lt;-- 0x100</td>
<td>%ebx = 0x100&lt;br&gt;state is set according to second irmovq instruction</td>
</tr>
<tr>
<td>Cycle 2:</td>
<td><code>irmovq $0x200,%rdx</code> # %rdx &lt;-- 0x200</td>
<td>%rdx = 0x200&lt;br&gt;combinational logic generates results for addq instruction</td>
</tr>
<tr>
<td>Cycle 3:</td>
<td><code>addq %rdx,%rbx</code> # %rbx &lt;-- 0x300, CC &lt;-- 000</td>
<td>%rbx = 0x300&lt;br&gt;CC = 000&lt;br&gt;results for addq instruction</td>
</tr>
<tr>
<td>Cycle 4:</td>
<td><code>je dest</code> # Not taken</td>
<td>Not taken&lt;br&gt;branch to destination</td>
</tr>
</tbody>
</table>

**CS429 Slideset 13: 21**

Datapath II
Cycle 1:
0x000: irmovq $0x100,%rbx # %rbx <-- 0x100

Cycle 2:
0x006: irmovq $0x200,%rdx # %rdx <-- 0x200

Cycle 3:
0x00c: addq %rdx,%rbx # %rbx <-- 0x300, CC <-- 000

Cycle 4:
0x00e: je dest # Not taken

- state is set according to addq instruction
- combinational logic starting to react to state changes
Cycle 1:
0x000: irmovq $0x100,%rbx # %rbx <-- 0x100

Cycle 2:
0x006: irmovq $0x200,%rdx # %rdx <-- 0x200

Cycle 3:
0x00c: addq %rdx,%rbx # %rbx <-- 0x300, CC <-- 000

Cycle 4:
0x00e: je dest # Not taken

- state is set according to addq instruction
- combinational logic generates results for je instruction
SEQ Summary

Implementation

- Express every instruction as a series of simple steps.
- Follow same general flow for each instruction type.
- Assemble registers, memories, predesigned combinational blocks.
- Connect with control logic.

Limitations

- Too slow to be practical. What is the slowest stage?
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory.
- Would need to run the clock very slowly.
- Hardware units are only active for a fraction of the clock cycle.
Where We’re Headed: Pipelining