CSC 252: Computer Organization
Fall 2021: Lecture 12

Processor Architecture:
Circuits
Single-cycle Implementation

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

Assignment 3 executable is 32-bit in order to support the modifications to modern Linux stack management

As a result, if you use gcc to generate code for this assignment, you will need to use “-m32” flag

You have about 2 weeks, but the midterm is right after, so make sure you have time for both

No class Monday: Fall break
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)

D
Data

C
Control

0 -> 1

Q+

Q−
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)
Edge-Triggered Latch (Flip-Flop)

- Flip-flop: Only latches data for a brief period
Edge-Triggered Latch (Flip-Flop)

- Flip-flop: Only latches data for a brief period
- Value latched depends on data as \( C \) rises (i.e., 0→1); usually called at the rising edge of \( C \)
Edge-Triggered Latch (Flip-Flop)

- Flip-flop: Only latches data for a brief period
- Value latched depends on data as \( C \) rises (i.e., 0\( \rightarrow \)1); usually called at the rising edge of \( C \)
- Output remains stable at all other times
Registers

- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal
Registers

- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal
Register Operation

State = x
Input = y  Output = x
Register Operation

State = x
Input = y  Output = x

C Rises
Register Operation

State = x
Input = y
Output = x

C Rises

State = y
Output = y

C
Register Operation

- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you’d better compute the input before the C signal rises if you want to store the input data to the register
Register Operation

- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you’d better compute the input before the C signal rises if you want to store the input data to the register

Output continuously produces y after the rising edge unless you cut off power.
Clock Signal

- A special C: periodically oscillating between 0 and 1
- That’s called the **clock** signal. Generated by a crystal oscillator inside your computer.
Clock Signal

- A special C: periodically oscillating between 0 and 1
- That’s called the **clock** signal. Generated by a crystal oscillator inside your computer.

```
Clock ________|________|________|________|________|________|________|
```

```plaintext
State = x
Input = y  Output = x

C Rises

State = y
Output = y
```
Clock Signal

- A special C: periodically oscillating between 0 and 1
- That’s called the **clock** signal. Generated by a crystal oscillator inside your computer.

```
State = x  Input = y  Output = x
  ^   ------  ^
  |   |     |  C Rises       State = y
  |   |     |              Output = y
  |   |     |              y
>C  
```

```
Clock

In  x_0 | x_1 | x_2 | x_3 | x_4 | x_5
```
Clock Signal

- A special C: periodically oscillating between 0 and 1
- That’s called the **clock** signal. Generated by a crystal oscillator inside your computer.
Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.
Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.
Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.
Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.
- 1 GHz CPU means the clock frequency is 1 GHz
Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.
- 1 GHz CPU means the clock frequency is 1 GHz
  - The cycle time is $1/10^9 = 1$ ns
Register File

- A register file consists of a set of registers that you can individually read from and write to.
Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out

![Register File Diagram]
Register File

• A register file consists of a set of registers that you can individually read from and write to.
• To read: give a register file ID, and read the stored value out
Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out.
Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value
Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out.
- To write: give a register file ID, a new value, overwrite the old value.
Register File

• A register file consists of a set of registers that you can individually read from and write to.
• To read: give a register file ID, and read the stored value out
• To write: give a register file ID, a new value, overwrite the old value
Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value
Register File

• A register file consists of a set of registers that you can individually read from and write to.
• To read: give a register file ID, and read the stored value out
• To write: give a register file ID, a new value, overwrite the old value
Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value
- How do we build a register file out of individual registers??
Register File Read

- Continuously read a register independent of the clock signal
Register File Read

- Continuously read a register independent of the clock signal
Register File Write

C0
Register 0

C1
Register 1

C2
Register 2

C3
Register 3

4:1 MUX

Read Reg ID

Out
Register File Write

Data

4:1 MUX

Register 0

Register 1

Register 2

Register 3

Read Reg ID

Out
Register File Write

Clock

Data

Register 0

Register 1

Register 2

Register 3

Read Reg ID

Out

4:1 MUX
Register File Write

• Only write the a specific register when the clock rises. How??

Clock

Write Reg ID

W1

W0

Data

Register 0

C0

D

Register 1

C1

D

Register 2

C2

D

Register 3

C3

D

4:1 MUX

Read Reg ID

Out
Register File Write

- Only write the a specific register when the clock rises. How??

![Diagram of Register File Write with data and control signals]
### Decoder

<table>
<thead>
<tr>
<th>W1</th>
<th>W0</th>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

W0 → W1 →

_C0
_C1
_C2
_C3
Decoder

<table>
<thead>
<tr>
<th>W1</th>
<th>W0</th>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

C0 = !W1 & !W0
C1 = !W1 & W0
C2 = W1 & !W0
C3 = W1 & W0
Decoder

<table>
<thead>
<tr>
<th>W1</th>
<th>W0</th>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

C0 = !W1 & !W0
C1 = !W1 & W0
C2 = W1 & !W0
C3 = W1 & W0
Register File Write
Register File Write

Clock

Write Reg ID

2:4 Decoder

Data

C0
Register 0

C1
Register 1

C2
Register 2

C3
Register 3

4:1 MUX

Read Reg ID

Out
Register File Write

Clock

Write Reg ID

Data

2:4 Decoder

Data

D

0

1

4:1MUX

Read Reg ID

Out

Register 0

C0

D

Register 1

C1

D

Register 2

C2

D

Register 3

C3

D
Register File Write

Clock

Write Reg ID

Data

2:4 Decoder

0

1

0

1

0

0

Register 0

Register 1

Register 2

Register 3

4:1 MUX

Read Reg ID

Out
Register File Write

Clock
Write Reg ID
Data

2:4 Decoder

Read Reg ID
Out

4:1 MUX

Register 0
Register 1
Register 2
Register 3

D
D
D
D

C0
C1
C2
C3
This implementation can read 1 register and write 1 register at the same time: 1 read port and 1 write port
Multi-Port Register File

- What if we want to read multiple registers at the same time?
Multi-Port Register File

- What if we want to read multiple registers at the same time?

![Diagram of a Multi-Port Register File](image-url)
Multi-Port Register File

- What if we want to read multiple registers at the same time?

- This register file has 2 read ports and 1 write port. How many ports do we actually need?
Multi-Port Register File

- Is this correct? What if we don’t want to write anything?
Multi-Port Register File

• Is this correct? What if we don’t want to write anything?
Processor Microarchitecture

• Sequential, single-cycle microarchitecture implementation
  • Basic idea
  • Hardware implementation

• Pipelined microarchitecture implementation
  • Basic Principles
  • Difficulties: Control Dependency
  • Difficulties: Data Dependency
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is `60 06`.

![Instruction Code and Function Code Diagram]

- **Instruction Code**
  - `addq rA, rB`
- **Function Code**
  - `6 0 rA rB`
Executing an ADD instruction

• How does the processor execute \texttt{addq \%rax,\%rsi}?
• The binary encoding is 60 06
Executing an ADD instruction

• How does the processor execute `addq %rax, %rsi`?
• The binary encoding is 60 06

```
Memory (Later…)
```

```
Clock
```

```
PC
```

```
Add
```

```
addq rA, rB 6 0 rA rB
```

```
Register File
```

```
Read Reg. ID 1
```

```
Enable
```

```
Clock
```

```
ALU
```

```
Flags
```

```
Z S O
```
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is 60 06.
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is 60 06
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is 60 06
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is `60 06`
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is 60 06

![Diagram of ADD instruction execution](image)
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is `60 06`
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is `60 06`
Executing an ADD instruction

- How does the processor execute \texttt{addq \%rax,\%rsi}?
- The binary encoding is 60 06
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is 60 06

![Diagram of the processor execution for the `addq %rax, %rsi` instruction.](image)
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is `60 06`.
Executing an ADD instruction

- How does the processor execute `addq %rax, %rsi`?
- The binary encoding is `60 06`
Executing an ADD instruction

- How does the processor execute \texttt{addq \%rax,\%rsi}
- The binary encoding is \texttt{60 06}
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
Executing an ADD instruction

• Logic 1: if (s0 == 6) select = s1;
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
Executing an ADD instruction

- Logic 1: if \((s0 == 6)\) select = s1;
- Logic 2: if \((s0 == 6)\) Enable = 1; else Enable = 0;

Logic 1: if \((s0 == 6)\) select = s1;
Logic 2: if \((s0 == 6)\) Enable = 1; else Enable = 0;

Logic 3

Memory (Later…)

Logic 2

Register File

Logic 1

ALU

Flags

Z

S

O
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

```
Logic 1: if (s0 == 6) select = s1;
Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
Logic 3: if (s0 == 6) nPC = oPC + 2;
Logic 4:
```
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?
Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?
Executing an ADD instruction

- When the rising edge of the clock arrives, the RF/PC/Flags will be written.
- So the following has to be ready: newData, nPC, which means Logic1, Logic2, Logic3, and Logic4 has to finish.
Executing a JLE instruction

- Let’s say the binary encoding for \texttt{jle .L0} is 71 0123000000000000
- What are the logics now?
Executing a JLE instruction

```
+---------+-----------------+-------------------+----------+
| Logic   | PC              | Memory            | Logic 1  |
| Logic 3 | nPC             |                   | ALU     |
| Memory  | s0 \[7\] s1 \[1\] s2 \[0\] s3 \[1\] s4 \[2\] s5 \[3\] \ldots \ldots |                 | Flags   |
| Logic 2 | oPC             |                   | Select  |
+---------+-----------------+-------------------+----------+

Logic 1: Select
ALU: Reg 1 Data, Reg 2 Data
Memory: newData
Register File: Write Reg. ID, Read Reg. ID 1, Read Reg. ID 2
Logic 3: Write Reg. ID
Logic 2: Enable, Clock
```

Note: The diagram illustrates the flow of data and control signals in executing a JLE instruction, including the interaction between registers, memory, and logic units.
Executing a JLE instruction

- Logic 1: if \( s0 == 6 \) select = \( s1 \);
Executing a JLE instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
Executing a JLE instruction

- Logic 3??
Executing a JLE instruction

- Logic 3??

if (s0 == 6) nPC = oPC + 2;
Executing a JLE instruction

- Logic 3??

```
if (s0 == 6) nPC = oPC + 2;
else if (s0 == 7) {
  if (s1 == 1) { // jLE
    if (Z || (S ^ O)) nPC = Dest; // jump
    else nPC = oPC + 10; // don’t jump, but add 10 (why??)
  } else if (s1 == …) {...
}
}
```
Executing a JLE instruction

- Logic 3??

```java
if (s0 == 6) nPC = oPC + 2;
else if (s0 == 7) {
    if (s1 == 1) { // jLE
        if (Z || (S ^ O)) nPC = Dest; // jump
        else nPC = oPC + 10; // don’t jump, but add 10 (why??)
    } else if (s1 == …) {…}
} else if (s1 == …) {…}
}
```
Executing a JLE instruction

- Logic 4? Does JLE write flags?
Executing a JLE instruction

- Logic 4? Does JLE write flags?
- Need another piece of logic.
Executing a JLE instruction

- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;
Microarchitecture (So far)
Microarchitecture (So far)

Clock -> PC -> Memory -> Register File -> Flags

Flags: Z, S, O
Microarchitecture (So far)

- Clock
  - PC
  - Memory
  - Register File
  - Flags (Z, S, O)
  - Inst.
Microarchitecture (So far)

Clock

PC \rightarrow Memory \rightarrow Register File \rightarrow Flags

Z S O

Combinational Logic
Microarchitecture (So far)

Clock

PC → Memory → Register File → Flags (Z S O)

Inst.

Combinational Logic

Read current_states;
Microarchitecture (So far)

Read `current_states;`
Microarchitecture (So far)

Read `current_states;`  
`next_states = f(current_states);`
Microarchitecture (So far)

Read current\_states;
next\_states = f(current\_states);
Microarchitecture (So far)

Clock

PC → Memory → Register File → Flags


Combinational Logic

Logic for generating ALU select signal
Logic for generating new flag value
Logic for generating new PC value
Logic for deciding all the enable signal values

Read current_states;
next_states = f(current_states);
Microarchitecture (So far)

Read `current_states`;
next_states = f(current_states);
When clock rises, current_states = next_states;
Microarchitecture (So far)

Clock

PC → Memory → Register File → Flags

Logic for generating ALU select signal
Logic for generating new flag value
Logic for generating new PC value
Logic for deciding all the enable signal values

Read current_states;
next_states = f(current_states);
When clock rises, current_states = next_states;
Executing a MOV instruction

- How do we modify the hardware to execute a move instruction?

```
rmmovq rA, D(rB)
```

```
move rA to the memory address rB + D
```

```
rmmovq %rsi,0x41c(%rsp)
```

```
40 64 1c 04 00 00 00 00 00 00
```

```
move rA to the memory address rB + D
```

```
rmmovq %rsi,0x41c(%rsp)
```

```
40 64 1c 04 00 00 00 00 00 00
```
move rA to the memory address rB + D

```
rmovq rA, D(rB)
```

- Register File
  - Write Reg. ID
  - Read Reg. ID 1
  - Read Reg. ID 2

- Logic 1
- Logic 2
- Logic 3
- Logic 4
- Logic 5

- Flags [s2…s9]
- Memory

- PC
- nPC
- oPC

- newData
move rA to the memory address rB + D

```
rmovq rA, D(rB)
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>rA</th>
<th>rB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>
```
move rA to the memory address rB + D

rmmovq rA, D(rB)
move rA to the memory address rB + D

rmmovq rA, D(rB)

| 4 | 0 | rA | rB | D |

• Need new logic (Logic 6) to select the input to the ALU for Enable.
move rA to the memory address rB + D

\[ rmmovq \ rA, \ D(rB) \]

- Need new logic (Logic 6) to select the input to the ALU for Enable.
move rA to the memory address rB + D

```plaintext
rmmovq rA, D(rB)
```

- Need new logic (Logic 6) to select the input to the ALU for Enable.
move \textit{rA} to the memory address \textit{rB + D}

\begin{center}
\begin{tabular}{c|c|c}
4 & 0 & \textit{rA} \text{ rB} \\
\end{tabular}
\end{center}

- Need new logic (Logic 6) to select the input to the ALU for Enable.
move rA to the memory address rB + D

\[
\text{rmmovq} \ rA, \ D(rB)
\]

- Need new logic (Logic 6) to select the input to the ALU for Enable.
- How about other logics?
How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?

move data at memory address rB + D to rA

\[
\text{mrmovq } D(rB), \ rA
\]

Data to write

Address

newData

Logic 6

Logic 1

Logic 2

Logic 3

Logic 4

Logic 5

How About Memory to Register MOV?
How About Memory to Register MOV?

move data at memory address rB + D to rA

`mrmovq D(rB), rA`
Microarchitecture (with MOV)

Combinational Logic

Read `current_states`;

```
next_states = f(current_states);
```

When clock rises, `current_states = next_states`;
Microarchitecture (with MOV)

Read current_states;
next_states = f(current_states);
When clock rises, current_states = next_states;
Microarchitecture (with MOV)

Read `current_states;`  
`next_states = f(current_states);`

When clock rises, `current_states = next_states;`
Microarchitecture (with MOV)

Read current_states;
next_states = f(current_states);
When clock rises, current_states = next_states;
Microarchitecture (with MOV)

Combinational Logic

Read `current_states`;
`next_states = f(current_states)`;
When clock rises, `current_states = next_states`;

`next_states` has to be ready before the close rises.