August 20, 2021, Tom’s hardware

**Tesla Packs 50 Billion Transistors Onto D1 Dojo Chip**

**Designed to Conquer Artificial Intelligence Training**

D1 delivers 362 TeraFLOPs of power.

Called the D1, the chip resembles a part of the Dojo supercomputer used to train AI models inside Tesla HQ, which are later deployed in various applications. The D1 chip is a product of TSMC’s manufacturing efforts, forged in a 7nm semiconductor node. Packing over 50 billion transistors, the chip boasts a huge die size of 645mm^2.

Image credit: Dennis Hong / Twitter
Review

• RISC-V ISA
  • Open, with increasing adoption

• RISC-V processor
  • A large state machine
  • Datapath + control
  • Reviewed R-, I-, S-format instructions and corresponding datapath elements
RISC-V
B-Format Instructions
Datapath So Far (R-, I-, S Instruction Types)

Control logic

Inst[31:0] ImmSel RegWEn Bsel ALUSel MemRW WBSel

Imm. Gen

IMEM

Inst

Add

Addr

Inst[31:7]

Inst[11:7]

Inst[19:15]

Inst[24:20]

+4

clk

Reg [ ]

Reg[rs1]

Reg[rs2]

DataD

AddrD

Imm[31:0]

DataA

AddrA

DataB

AddrB

+ ALU

Bsel

ALUSel

MemRW

WBSel

DataR

DataW

mem

wb

pc+4

pc

+4

Add

addr

inst

IMEM

IMEM

Inst[31:0]

ImmSel

RegWEn

Nikolić, Fall 2021

EECS151 L09 PIPELINING
B-Format - RISC-V Conditional Branches

• E.g., \texttt{BEQ x1, x2, Label}
• Branches read two registers but don’t write a register (similar to stores)
• How to encode label, i.e., where to branch to?
RISC-V Feature, n×16-bit instructions

• Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length

• To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions

• Reduces branch reach by half and means that ½ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)

• RISC-V conditional branches can only reach ± $2^{10} \times 32$-bit instructions on either side of PC
RISC-V Branch Immediates

- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- RISC-V approach: keep 11 immediate bits in fixed position in output value, and rotate LSB of S-format to be bit 12 of B-format

### S-Immediate

- `s imm[10:5] rs2 rs1 funct3 imm[4:0] B-opcode`

### B-Immediate (shift left by 1)


Only one bit changes position between S and B, so only need a single-bit 2-way mux
RISC-V Immediate Encoding

Instruction encodings, inst[31:0]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit immediates produced, imm[31:0]

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B
beq  x19, x10, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16
imm[0] discarded, always zero
imm[12]


Nikolić, Fall 2021
Implementing Branches

• B-format is mostly same as S-format, with two register sources (rs1/rs2) and a 12-bit immediate

• But now immediate represents values -4096 to +4094 in 2-byte increments

• The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
To Add Branches

• Different change to the state:
  • $PC = \begin{cases} 
  PC + 4, & \text{branch not taken} \\
  PC + \text{immediate}, & \text{branch taken}
\end{cases}$

• Six branch instructions: BEQ, BNE, BLT, BGE, BLTU, BGEU

• Need to compute $PC + \text{immediate}$ and to compare values of $rs1$ and $rs2$
  • Need another add/sub unit
Adding Branches

PCSel = taken/not taken

Control logic

Inst[31:0] ImmSel = B RegWEn = 0 BrUn = 1 BrLT = 1 BrEq = 1 ALUSel = Add = 1 MemRW = Read WBSel = * (=Don't care)

Inst[11:7] AddrD AddrA DataA


Imm[31:0]

Imm. Gen

Reg [ ]

Reg[rs1]

Branch Comp

Branch

Comp

Clk

DataR

DataW

DMEM

Mem

wb

alu

ALU

+4

Add

WBSel = * (=Don't care)

Reg[rs1]

Reg[rs2]

alu

RegWEn = 0

ALUSel = Add = 1

MemRW = Read

WBSel = * (=Don't care)

ImmSel = B

BrUn = 1

BrLT = 1

BrEq = 1

ALUSel = Add = 1

MemRW = Read

WBSel = * (=Don't care)

ImmSel = B

BrUn = 1

BrLT = 1

BrEq = 1

ALUSel = Add = 1

MemRW = Read

WBSel = * (=Don't care)
Branch Comparator

- \( \text{BrEq} = 1 \), if \( A = B \)
- \( \text{BrLT} = 1 \), if \( A < B \)
- \( \text{BrUn} = 1 \) selects unsigned comparison for \( \text{BrLT} \), 0 = signed

- BGE branch: \( A \geq B \), if \( A < B \)
### All RISC-V Branch Instructions

|-----------|-----|-----|-----|-----------|---------|

- **BEQ**
- **BNE**
- **BLT**
- **BGE**
- **BLTU**
- **BGEU**
Administrivia

• Homework 4 is due next Monday
  • No new homework this week
  • Homework 5 will be posted next week, due after the midterm

• Lab 5 this week
  • No lab next week
  • Lab 6 (last) after the midterm

• Midterm 1 on October 7, 7-8:30pm
RISC-V

J-Format Instructions
J-Format for Jump Instructions

- JAL saves PC+4 in register rd (the return address)
  - Assembler “j” jump is pseudo-instruction, uses JAL but sets \( rd = x0 \) to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within \( \pm 2^{19} \) locations, 2 bytes apart
  - \( \pm 2^{18} \) 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

offset[20:1] dest JAL
JALR Instruction (I-Format)

- JALR rd, rs, immediate
  - Writes PC+4 to rd (return address)
  - Sets PC = rs1 + immediate (and sets the LSB to 0)
  - Uses same immediates as arithmetic and loads
    - no multiplication by 2 bytes
    - In contrast to branches and JAL

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>func3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>0</td>
<td>dest</td>
<td>JALR</td>
</tr>
</tbody>
</table>

Otherwise, we would have yet another new encoding
Let’s Add JALR (I-Format)

- JALR rd, rs, immediate

- Two changes to the state
  - Writes PC+4 to rd (return address)
  - Sets PC = rs + immediate
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
    - LSB is ignored
Datapath So Far, with Branches
Adding JALR

Control logic:
- PCSel = taken
- ImmSel = I
- RegWEn = 1
- BrUn = *
- BrEq = *
- Bsel = 1
- AlUSel = Add
- MemRW = Read
- WBSel = 2

IMEM:
- pc + 4
- Inst[24:20]
- AddrB
- DataB
- Reg[rs1]
- clk

Inst[31:7] = Imm[31:0]

Add:
- Add

Branch Comp:
- + ALU
- Branch
- Comp

DataD:
- DataR
- DataW

DMEM:
- mem
- wb

Reg [ ]:
- Reg[rs2]

alu:
- +
- ALU

Imm Gen:
- Imm[31:0]

ALUSel = Add
MemRW = Read
WB Sel = 2

Nikolić, Fall 2021
EECS151 L09 PIPELINING
Adding **JAL**

- JAL saves PC+4 in register rd (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within ±2^{19} locations, 2 bytes apart
  - ±2^{18} 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
Adding JAL

Control logic:
- PCSel = taken
- Inst[31:0]
- ImmSel = J
- RegWEn = 1
- BrUn = *
- BrLT = *
- Bsel = 1
- ALUSel = add
- MemRW = Read
- WBSel = 2
RISC-V
U-Format Instructions
U-Format for “Upper Immediate” Instructions

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - `lui` – Load Upper Immediate
  - `auipc` – Add Upper Immediate to PC
Implementing lui

1. **IMEM**: IMEM is used to fetch the instruction from memory.
2. **Addr**: Address computation is performed here.
3. **Add**: Adder performs the addition of the address.
4. **Reg**: Registers store the results of the ALU operations.
5. **IMU**: Instruction Memory Unit fetches the instructions.
6. **DMEM**: Data Memory Unit stores and retrieves data.
7. **ALU**: Arithmetic Logic Unit performs arithmetic and logical operations.
8. **Branch Comp**: Branch comparator checks for branch conditions.
9. **WBSel**: Write Buffer Selection
10. **MemRW**: Memory Read/Write selection
11. **PCSel**: Program Counter Selection
12. **Inst[31:0]**: Instruction bus
13. **ImmSel**: Immediate Selection
14. **RegWEn**: Register Write Enable
15. **BrUn**: Branch Unconditional
16. **BrLT**: Branch Less Than
17. **BrEq**: Branch Equal
18. **Bsel**: Branch Selection
19. **ALUSel**: ALU Selection
20. **MemRW**: Memory Read/Write
21. **WBSel**: Write Buffer Selection

The diagram shows the flow of data and control signals through the pipeline stages.
Implementing auipc
Complete RV32I Datapath!
Recap: Complete RV32I ISA

- 40 instructions are enough to run any C program
Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Index</th>
<th>Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>funct7</td>
<td>rs2</td>
</tr>
<tr>
<td>30</td>
<td>rs1</td>
<td>funct3</td>
</tr>
<tr>
<td>29</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td>28</td>
<td>imm[11:0]</td>
<td>rs1</td>
</tr>
<tr>
<td>27</td>
<td>funct3</td>
<td>imm[4:0]</td>
</tr>
<tr>
<td>26</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td>24</td>
<td>rs1</td>
<td>funct3</td>
</tr>
<tr>
<td>22</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>imm[12:10:5]</td>
<td>rs2</td>
</tr>
<tr>
<td>20</td>
<td>rs1</td>
<td>funct3</td>
</tr>
<tr>
<td>18</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>imm[31:12]</td>
<td>rd</td>
</tr>
<tr>
<td>16</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

R-type: Index 0, opcode imm[31:12]
I-type: Index 0, imm[11:0] rs1 funct3 imm[4:0] opcode
Control and Status Registers (CSRs)

- 4096 CSRs in a separate address space

<table>
<thead>
<tr>
<th>csr</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

- csrrw reads the old value of CSR, zero-extends and writes to rd
- Initial value of rs1 is written to CSR
- Pseudo-instructions: csrr, csrw (csrrw x0, csr, rs1)
Add the **csrrw**!
RISC-V Control Logic
Complete RV32I Datapath with Control

[Nikolić, Fall 2021]

EECS151 L09 PIPELINING

Nikolić, Fall 2021 36
Example: add

PCSel = pc+4

Inst[31:0]

ImmSel = *

RegWEn = 1

BrUn = *

BrLT = *

Bsel = 0

Asel = 0

ALUSel = add

MemRW = Read

LdSel = WBSel = 1

Control logic

funct7  rs2  rs1  funct3  rd  opcode

7  5  5  3  7  5  25  24  20  19  15  14  12  11  7  6  0

Nikolić, Fall 2021

EECS151 L09 PIPELINING
add Execution

Clock
PC

PC+4

inst[31:0]

Control logic

Reg[rs1]

Reg[rs2]

alu

wb

Real[11]

add x1, x2, x3

add x6, x7, x9

add control


add control

Reg[2]

Reg[3]

Reg[7]

Reg[9]


## Control Logic Truth Table

<table>
<thead>
<tr>
<th>Inst[31:0]</th>
<th>BrEq</th>
<th>BrLT</th>
<th>PCSel</th>
<th>ImmSel</th>
<th>BrUn</th>
<th>ASel</th>
<th>BSel</th>
<th>ALUSel</th>
<th>MemRW</th>
<th>RegWEn</th>
<th>WBSel</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>sub</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>Sub</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>(R–R Op)</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>(Op)</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>addi</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>lw</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>Mem</td>
</tr>
<tr>
<td>sw</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>S</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Write</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>*</td>
<td>+4</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>*</td>
<td>ALU</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bne</td>
<td>0</td>
<td>*</td>
<td>ALU</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bne</td>
<td>1</td>
<td>*</td>
<td>+4</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>blt</td>
<td>*</td>
<td>1</td>
<td>ALU</td>
<td>B</td>
<td>0</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bltu</td>
<td>*</td>
<td>1</td>
<td>ALU</td>
<td>B</td>
<td>1</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>jalr</td>
<td>*</td>
<td>*</td>
<td>ALU</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>PC+4</td>
</tr>
<tr>
<td>jal</td>
<td>*</td>
<td>*</td>
<td>ALU</td>
<td>J</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>PC+4</td>
</tr>
<tr>
<td>auipc</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>U</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
</tbody>
</table>
RV32I, a nine-bit ISA!

Instruction type encoded by using only 9 bits: \text{inst}[30], \text{inst}[14:12], \text{inst}[6:2]
Control Realization Options

• **ROM**
  - “Read-Only Memory”
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions

• **Combinatorial Logic**
  - Start from a truth table
  - More compact, faster
  - Use synthesis tools

```
inst[30], inst[14:12], inst[6:2]
```
Combinational Logic Control

• Decoder is typically hierarchical
  • First decode opcode, and figure out instruction type
  • E.g. branches are Inst[6:2] = 11000
  • Then determine the actual instruction
    • Inst[30] + Inst[14:12]

• Modularity helps simplify and speed up logic
  • Narrows problem space for logic synthesis
Combinational Logic Control

• Simple example: BrUn

```
inst[14:12]  inst[6:2]
```

• How to decode whether BrUn is 1?

  • BrUn = Inst [13] • Branch

Critical path for a addi

\[ R[rd] = R[rs1] + \text{imm} \]

1) \( t_{\text{clk-q}} + t_{\text{Add}} + t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{BComp}} + t_{\text{ALU}} + t_{\text{DMEM}} + t_{\text{mux}} + t_{\text{Setup}} \)

2) \( t_{\text{clk-q}} + t_{\text{IMEM}} + \max\{t_{\text{Reg}}, t_{\text{Imm}}\} + t_{\text{ALU}} + 2t_{\text{mux}} + t_{\text{Setup}} \)

3) \( t_{\text{clk-q}} + t_{\text{IMEM}} + \max\{t_{\text{Reg}}, t_{\text{Imm}}\} + t_{\text{ALU}} + 3t_{\text{mux}} + t_{\text{DMEM}} + t_{\text{Setup}} \)

4) None of the above
Peer Instruction(s): Critical Path yellkey: physical

Critical path for a addi

\[ R[rd] = R[rs1] + \text{imm} \]

1) \[ t_{\text{clk-q}} + t_{\text{Add}} + t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{BComp}} + t_{\text{ALU}} + t_{\text{DMEM}} + t_{\text{mux}} + t_{\text{Setup}} \]

2) \[ t_{\text{clk-q}} + t_{\text{IMEM}} + \max\{t_{\text{Reg}}, t_{\text{Imm}}\} + t_{\text{ALU}} + 2t_{\text{mux}} + t_{\text{Setup}} \]

3) \[ t_{\text{clk-q}} + t_{\text{IMEM}} + \max\{t_{\text{Reg}}, t_{\text{Imm}}\} + t_{\text{ALU}} + 3t_{\text{mux}} + t_{\text{DMEM}} + t_{\text{Setup}} \]

4) None of the above

Nikolić, Fall 2021
Pipelining
Complete RV32I Datapath with Control

Instruction Fetch (F)

Instruction Decode/Register Read (D)

ALU Execute (X)

Memory Access (M)

Write Back (W)

1. **PC**
   - +4
   - Addr
   - Inst

2. **IMEM**
   - pc+4
   - addr
   - inst
   - IMEM

3. **AddrB**
   - AddrB
   - DataB

4. **DataA**
   - DataA
   - Inst[11:7]
   - AddrA
   - Inst[19:15]
   - AddrD
   - Inst[24:20]
   - Addr
   - Inst[31:7]
   - Imm[31:0]
   - Imm.
   - Gen
   - Addr
   - Inst[31:7]
   - Imm[31:0]

5. **alu**
   - Reg[rs1]
   - Reg[rs2]

6. **Branch Comp**
   - Branch
   - Comp

7. **ALU**
   - +
   - clk
   - Reg[rs1]
   - Reg[rs2]

8. **DataR**
   - DataR
   - LD
   - X
   - mem

9. **DMEM**
   - DMEM
   - mem
   - clk

10. **Reg[ ]**
    - Reg[ ]
    - pc
    - pc+4
    - wb
    - wb

11. **Write Back**
    - Write
    - Back

Nikolić, Fall 2021
Pipelining RV32I Datapath

Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline.

Instruction Fetch (F)

Instruction Decode/Register Read (D) - instruction along with data

ALU Execute (X)

Memory Access (M)

Write Back (W)

Must pipeline instruction along with data, so control operates correctly in each stage.
Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline
Different Instructions in Flight

lw \text{t0, 8(t3)}

sw \text{t0, 4(t3)}

\text{slt t6, t0, t3}

or \text{t3, t4, t5}

add \text{t0, t1, t2}

\text{Instruction Fetch (F)}

\text{Instruction Decode/Register Read (D)}

\text{ALU Execute (EX)}

\text{Memory Access (M)}

\text{Write Back (WB)}

\text{Imm. Gen [31:0]}

\text{Reg [rs1]}

\text{InstM}

\text{InstX}

\text{InstD}

\text{AddrD}

\text{AddrA}

\text{AddrB}

\text{DataA}

\text{DataB}

\text{DataD}

\text{Reg [ ]}

\text{pcx}

\text{wb}

\text{alu}

\text{clk}

\text{addr}

\text{inst}

\text{IMEM}

\text{DMEM}

\text{LD}

\text{X}

\text{PC}

Nikolić, Fall 2021
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages
Summary

• RISC-V ISA
  • Completed the datapath with B-, J-, U-instructions

• Control
  • Can be implemented as a ROM while prototyping
  • Synthesized as custom logic

• Pipelining to increase throughput
  • 5-stage pipeline example