Computer Architecture
Processor implementation

http://d3s.mff.cuni.cz/teaching/nswi143

Lubomír Bulej
bulej@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE
faculty of mathematics and physics
Implementing simplified MIPS ISA

- **Basic characteristics**
  - Simplified to demonstrate key concepts

- **Registers**
  - 32 general-purpose 32-bit registers: R0 – R31
  - PC registers with address of instruction to execute
  - Special control registers
    - Exception address, etc.
Implementing simplified MIPS ISA (2)

- **Memory**
  - Access to 4-byte aligned addresses only
    - Corresponds to 32-bit word length of the processor
  - Indirect addressing with immediate displacement
    - **Load:** R2 := mem[R1 + immediate]
    - **Store:** mem[R1 + immediate] := R2
Implementing simplified MIPS ISA (3)

- **Operations**
  - **Arithmetic and logic**
    - Fully orthogonal, three-operand instructions
    - Source operands: register/register, register/immediate
    - Target operand: register
    - Includes data movement between registers
  - **Load/store operations**
    - Move data between registers and memory (load/store architecture)
  - **Conditional branch**
    - Tests equality/inequality of two registers
  - **Unconditional jumps**
    - Including jumps to subroutine and indirect jumps (return from a subroutine)
  - **Special instructions**
Implementing simplified MIPS ISA (4)

- **Single-cycle datapath**
  - Basic organization of data path elements
    - Combinational and sequential blocks
  - Operations executed in one long cycle
    - Suitable for operations of similar complexity
    - Writes to memory elements synchronized by clock
      - Clock signal is implicit, will not be shown
  - **Simplification**: separate instruction memory (Harvard architecture)
Implementing simplified MIPS ISA (5)

- **Steps to execute an instruction**

  1. Fetch instruction from memory
     - Read from an address supplied by the PC register
  2. Decode instruction and fetch instruction operands
  3. Execute operation corresponding to the opcode
     - Register operations, computing address for accessing memory, comparing operands for conditional branch.
  4. Store the result of the operation
     - Write data to register or memory
  5. Adjust PC to point at next instruction
     - One that immediately follows the current
     - One that is a target of a jump or branch
Reading an instruction (fetch)

- **PC register**
  - Address of instruction in memory
  - Not directly accessible to a programmer

- **Adder**
  - Increment PC by 4
  - Advance to next instruction by default
Register operations (add, sub, ...)

<table>
<thead>
<tr>
<th>op (6)</th>
<th>rs (5)</th>
<th>rt (5)</th>
<th>rd (5)</th>
<th>sa (5)</th>
<th>funct (6)</th>
</tr>
</thead>
</table>

- **RegWrite**
  - Read register 1
  - Read register 2
  - Write register
  - Write data

- **ALUOp**
  - funct
  - ALU
  - result

- **Register File**
  - 32
  - 32
  - 32

- **OP**
  - 5
  - 5
  - 5
  - 5
Support for register operations

4

PC

Addr

Insn

IM

RS

RT

RD

WD

RF

I[25:21]

I[20:16]

I[15:11]

32

32

32

ALU

RegWrite

ALUOp

result

add

4
Immediate operand operations (addi, ...)
Implementing sign extension

Sign Extend 16 to 32 bits

$\begin{align*}
  x_0 & \rightarrow y_0 \\
\vdots & \vdots \\
  x_{15} & \rightarrow y_{16} \\
  \rightarrow & y_{15} \\
  \rightarrow & y_{16} \\
  \rightarrow & y_{31}
\end{align*}$
Support for immediate operands

PC → Addr → IM → add → Addr → IM

I[15:0] → Sign ext. → [15:0]

I[25:21] → [25:21]
I[20:16] → [20:16]
I[15:11] → [15:11]

ALUOp → ALU → result

RegWrite → RS → A
RS → A
ALUSrc → MUX → 32
ALUSrc → MUX → 32

PC → Addr → IM

4 → add

Insn → Addr → IM

RegDst → MUX → 32

RD → MUX → 32

RF → RD

Computer Architecture, Processor implementation, summer 2020/2021
Multiplexer (mux)

- Selects one of several inputs
  - **Selector:** $n$-bit number $S \in \{0, ..., 2^{n-1}\}$
  - **Data input:** $N=2^n$ $m$-bit values $x_0, x_1, ..., x_{N-1}$
  - **Data output:** $m$-bit value $y=x_S$

![Multiplexer Diagram](image-url)
Implementing a multiplexer

- **Binary to “1-hot” decoder**
  - Activates 1 (selected output) of N outputs
  - **Input:** $n$-bit number $B \in \{0, \ldots, 2^{n-1}\}$
  - **$N=2^n$ outputs:** $B$-th output logical 1 (hot), other outputs logical 0
Binary to 1-hot for $N=4$ outputs

<table>
<thead>
<tr>
<th>Inputs $B_1 B_0$</th>
<th>Outputs $h_3 h_2 h_1 h_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

1-hot

$B_0$ $B_1$

$h_0$ $h_1$ $h_2$ $h_3$
Implementing a multiplexer (4x 1-bit)

Mux

1-hot

$S$

$x_0$

$x_1$

$x_2$

$x_3$

$y$
Loading words from memory (lw)

```
# op (6)  # rs (5)  # rt (5)  # displacement (16)
```

```
displacement

16  

Sign ext.

RegWrite

ALUOp

rs

5

Read register 1

Write register

Write data

rt

5

Register data 1

32

ALU

Address

Data

Data Memory

32

32

A

L

U

Address

Data

Memory

5

5

32

Register File

32

22

16

Sign ext.

16

32

32

Address

Data

Data Memory

32

32

Sign ext.

16

32

RegWrite

ALUOp

rs

5

Read register 1

Write register

Write data

rt

5

Register data 1

32

ALU

Address

Data

Data Memory

32

32

A

L

U

Address

Data

Memory

5

5

32

Register File

32

22

16

Sign ext.

16

32

RegWrite

ALUOp

```
Storing words to memory (sw)

```
|  op (6)  |   rs (5)   |    rt (5)   | displacement (16) |
```

```
<table>
<thead>
<tr>
<th>displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>RegWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp</td>
</tr>
<tr>
<td>MemWrite</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Read register 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read register 2</td>
</tr>
<tr>
<td>Register data 1</td>
</tr>
<tr>
<td>Register data 2</td>
</tr>
</tbody>
</table>
```

```
| 32 |
| 32 |
| 32 |
| 32 |
```

```
<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Data Memory</td>
</tr>
</tbody>
</table>
```

```
| 5 |
| 5 |
```

```
| Sign ext. |
```

```
| rs (5) |
| rt (5) |
```

```
| ALU |
```

```
| A |
| L |
| U |
```

```
| 16 |
```
Support for memory access (load/store)
Conditional branch relative to PC (beq)

- **op**: 6
- **rs**: 5
- **rt**: 5
- **offset**: 16

Diagram:
- **PC**: 4
- **add**:
  - **MUX**:
  - **offset**: 16
  - **Sign ext.**: 32
  - **Shl 2**: 32
  - **RegWrite**:
    - **ALUOp**:
      - **ALU**:
        - **zero?**:
          - **Read register 1**:
            - **Register data 1**: 32
          - **Read register 2**:
            - **Register data 2**: 32
          - **Register File**:
            - **32**

**Branch**
Implementing logical shift

32-bit shift left logical 2

\( x_{31} \rightarrow y_{31} \)
\( x_{30} \rightarrow \ldots \)
\( x_{29} \rightarrow y_3 \)
\( \vdots \)
\( x_1 \rightarrow y_2 \)
\( x_0 \rightarrow y_1 \)
\( \ldots \)
\( 0 \rightarrow y_0 \)
Support for conditional branch

Diagram showing the support for conditional branch with various registers and signals such as ALUOp, RegWrite, MemWrite, and Branch.
Unconditional jump (j)

Unconditional jump (j)

op (6) target (26)

PC → add

PC + 4[31:28] → 4

Shift (and extend) left by 2

32 → 28 → 26

Sxl 2

target
Support for unconditional jump
Single-cycle datapath control
Single-cycle datapath control (2)

- **Controls the flow of data**
  - Depending on the type of operation
  - Responsible for control signals
    - Source of the next value of PC
    - Write to registers
    - Write to memory
    - ALU operations
    - Mux configuration
Example: datapath control for *add*

- **Jump** = 0
- **Add**
- **PC**
- **Addr**
- **Insn**
- **IM**
- **ALUOp** = *add*
- **RegWrite** = 1
- **ALUSrc** = 1
- **RegDst** = 1
- **MemToReg** = 0
- **MemWrite** = 0
- **Branch** = 0
Example: datapath control for sw

Jump=0

Branch=0

ALUOp=add

MemWrite=1

RegWrite=0

ALUSrc=0

MemToReg=?
Example: datapath control for `beq`

- **Jump=0**
- **Branch=1**
- **RegWrite=0**
- **ALUOp=sub**
- **MemWrite=0**
- **ALUSrc=1**
- **MemToReg=1**
- **RegDst=?**

Diagram showing the control flow for the `beq` instruction, including ALU operations, registers, and memory access.
Datapath controller

- **Responsible for generating control signals**
  - Signal values determined by instruction opcode
  - Some control signals can be directly embedded in the instruction word
    - **MIPS**: $ALUOp$ signals correspond to the bits in the `funct` field of the R-type instruction format
    - Simplifies controller implementation
ROM-based controller

- **Signal values stored in read-only memory**
  - Each word contains the values of all control signals
  - Words addressed by the opcode

<table>
<thead>
<tr>
<th>opcode</th>
<th>Jump</th>
<th>Branch</th>
<th>RegDst</th>
<th>RegWrite</th>
<th>MemWrite</th>
<th>MemToReg</th>
<th>ALUOp</th>
<th>ALUSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>func</td>
<td>1</td>
</tr>
<tr>
<td>addi</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>add</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>add</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>sub</td>
<td>1</td>
</tr>
</tbody>
</table>
ROM-based controller (2)

- **Real MIPS implementation**
  - Approx. 100 instructions and 300 control signals
    - Control ROM capacity needed: 30000 bits (~ 4 KB)
  - Implementation issues
    - Making ROM faster than the datapath
Logic-based controller (combinational)

- Faster alternative to ROM
  - Observation: only a few control signals need to be set to one (zero) at the same time
  - Contents of ROM can be efficiently expressed using logic functions
**Datapath with continuous read**

- No problem in our design
  - Writes (PC, RF, DM) are independent
  - No read follows write in the instruction cycle
  - Instruction fetch does not need control
    - After instruction is read, the controller decodes instruction opcode into control signals for the rest of the datapath
    - When PC changes, datapath starts processing another instruction

---

**Diagram:**

- Read from insn memory
- Read registers (Read control ROM)
- Read from data memory
- Write to data memory
- Write to registers
- Write to PC
Each instruction executed in 1 cycle (CPI=1)

- Single-cycle controller (control ROM or a combinational logic block)
- Generally lower clock frequency
- Clock period respects the “longest” instruction
  - Load Word (lw) in our case
  - Usually multiplication, division, or floating point ops
- Datapath contains duplicate elements
  - Instruction and data memory, two extra adders
Multi-cycle datapath

- **Basic idea**
  - Simple instructions should not take as much time to execute as the complex ones

- **Variable instruction execution time**
  - Clock period is constant (cannot be changed dynamically), we need a „digital“ solution
  - We can make clock faster (shorter period) and split instruction execution into multiple stages
    - Clock period corresponds to **one execution stage**
    - Fixed **machine cycle** (clock period)
    - Variable **instruction cycle**
Example: multi-cycle CPU performance

- **Rough estimate, assuming the following**
  - Simple instructions take 10 ns to execute
  - Multiplication takes 40 ns
  - Instruction mix with 10% of multiplications

- **Single-cycle datapath**
  - Clock period 40 ns, CPI=1 $\rightarrow$ **25 MIPS**

- **Multi-cycle datapath**
  - Clock period 10 ns, 13 ns per instruction (average)
  - CPI=1.3 $\rightarrow$ **77 MIPS** (3x improvement)
Multi-cycle datapath (2)

Instruction cycle

1. Read instruction from memory
2. Decode instruction, read registers, compute branch target address
3. Execute register operation / compute address for memory access / finish branch or jump
4. Write register operation results / access memory
5. Finish load from memory
Multi-cycle datapath (3)

- Implementation issues
  - Instruction execution split to stages
    - Need to isolate stages using latch registers to “remember” results from previous stage
  - Need to keep track of stages
    - Different sequences for different instruction types
    - Some instructions may skip stages and finish early
    - Controller needs to remember state → sequential logic
Multi-cycle datapath (4)
Stage 1: Instruction Read

- **Common for all instructions**
  - IR \(\leftarrow\) Memory[PC]
    - Read instruction into Instruction Register
    - Memory is used for both instruction and data access
    - Need to “remember” the instruction being executed
  - PC \(\leftarrow\) PC + 4
    - Advance PC to point at next instruction in sequence
    - Changing the PC will not change the instruction being executed: it was stored in the Instruction Register
Stage 2: Instruction Decode, Read Regs.

- **Common for all instructions**
  - A ← Reg[IR.rs]
    - Read contents of source register 1
    - Store value into latch A for next stage
  - B ← Reg[IR.rt]
    - Read contents of source register 2
    - Store value into latch B for next stage
  - ALUOut ← PC + (SignExtend(IR.addr) << 2)
    - Calculate branch target
    - Relative to (already updated) PC
    - Remains unused if not a branch
Stage 3: Execute / address calc.

- **Branch instruction (finish)**
  - \((A == B) \Rightarrow PC \leftarrow ALUOut\)
    - Branch target in ALUOut from previous stage

- **Jump instruction (finish)**
  - \(PC \leftarrow PC[31:28] + (IR[25:0] \ll 2)\)

- **Register operation**
  - \(ALUOut \leftarrow A\) *funct* \(B\), or alternatively
  - \(ALUOut \leftarrow A\) *funct* \(\text{SignExtend}(IR[15:0])\)

- **Memory access**
  - \(ALUOut \leftarrow A + \text{SignExtend}(IR[15:0])\)
    - Calculate address for memory access
Stage 4: Write Results / memory access

- **Register operation** *(finish)*
  - Reg[IR.rd] ← ALUOut
    - Result in ALUOut (from previous stage)

- **Write to memory** *(finish)*
  - Memory[ALUOut] ← B
    - Address in ALUOut (from previous stage)

- **Read from memory**
  - DR ← Memory[ALUOut]
    - Address in ALUOut (from previous stage)
    - Store data into latch DR for next stage
Stage 5: Finish reading from memory

- **Read from memory (**finish**)**
  - Reg[IR.rt] ← DR
    - Value stored in DR (from previous stage)
Multi-cycle datapath implementation

PC

IorD MemWrite

Insn Register

[25:21]
[20:16]
[15:0]

IRWrite RegDst RegWrite

Read register 1
Read register 2
Write register
Write data

Register File

ALUSrcA

ALUSrcB

ALU

ALUOp

ALU Out

Register

data

1 0

[15:11]

MemRead

Data Register

[15:0]

MemToReg

[15:0] 16 Sign ext. 32

[5:0]

Shl 2
Multi-cycle datapath control

- **Sequential process**
  - Instructions executed in multiple cycles
  - Controller is a sequential circuit (automaton)
    - Current state stored in a state register
    - Combinational block determines next state
      - Depends on current state and instruction being executed
      - Updated on rising edge of the clock signal

![Diagram of multi-cycle datapath control]

START

- Instruction fetch/decode
  - Register fetch

- Memory access instructions
- R-type instructions
- Branch instruction
- Jump instruction
Instruction fetch/decode, Register fetch

Instruction fetch

START

0

MemRead
ALUSrcA=0
IorD=0
IRWrite
ALUSrcB=01
ALUOp=00
PCWrite
PCSource=00

Instruction decode
Register fetch

1

ALUSrcA=0
ALUSrcB=11
ALUOp=00

Op=='lw'
Op is R-type
Op=='sw'
Op='beq'
Op='j'

Memory access instructions

R-type instructions

Branch instruction

Jump instruction

Instruction fetch/decode, Register fetch
Memory access instructions

Memory address computation

1. \(\text{ALUSrcA}=1\)
   \(\text{ALUSrcB}=10\)
   \(\text{ALUOp}=00\)

2. \(\text{Op}==\text{lw}'\)

3. Memory access
   \(\text{MemRead}\)
   \(\text{IorD}=1\)

4. Memory read completion step
   \(\text{RegWrite}\)
   \(\text{MemToReg}=1\)
   \(\text{RegDst}=0\)

5. \(\text{Op}==\text{sw}'\)

   Memory access
   \(\text{MemWrite}\)
   \(\text{IorD}=1\)

To 0
R-type instructions

R-type execution

6

From 1

ALUSrcA=1
ALUSrcB=00
ALUOp=10

7

R-type completion

RegDst=1
MemToReg=0
RegWrite=1

To 0
Branch instruction

Branch completion

From 1

ALUSrcA=1
ALUSrcB=00
ALUOp=01
PCWriteCond
PCSource=01

12

To 0
Jump instruction

Jump execution

From 1
PCWrite
PCSource=10

To 0

13
Multi-cycle datapath control (2)

Instruction fetch
PC update

START

MemRead
ALUSrcA=0
IorD=0
IRWrite
ALUSrcB=01
ALUOp=00
PCWrite
PCSource=00

MemRead
ALUSrcA=1
ALUSrcB=10
ALUOp=00

ALUSrcA=1
ALUSrcB=00
ALUOp=10

RegDst=1
MemToReg=0
MemWrite
IorD=1

RegWrite
MemToReg=1
RegDst=0

Instruction decode
Register fetch
Branch target

PCWrite
PCSource=10

Jump execution

ALUSrcA=1
ALUSrcB=00
ALUOp=01
PCWriteCond
PCSource=01

Branch completion

Op==lw || Op==sw

Op==r-type

Op==j

Op==beq

R-type execution

R-type completion

Memory address computation

Memory access

Memory load completion

0

1

2

3

4

5

6

7
Addi instruction

Addi execution

From 1

ALUSrcA=1
ALUSrcB=10
ALUOp=00

To 0

I-type completion

RegDst=0
MemToReg=0
RegWrite=1
Multi-cycle datapath control (3)

Instruction fetch
- PC update

Instruction decode
- Register fetch
- Branch target

ALUSrcA=0
- ALUSrcB=01
- ALUOp=00
- PCWrite

PCSource=00

ALUSrcA=0
- ALUSrcB=11
- ALUOp=00

START

MemRead
- ALUSrcA=0
- IorD=0
- IRWrite
- ALUSrcB=01
- ALUOp=00
- PCWrite

PCSource=00

Op==lw || Op==sw

ALUSrcA=1
- ALUSrcB=10
- ALUOp=00

ALUSrcA=1
- ALUSrcB=00
- ALUOp=10

ALUSrcA=1
- ALUSrcB=10
- ALUOp=00

ALUSrcA=1
- ALUSrcB=00
- ALUOp=01

ALUSrcA=1
- ALUSrcB=00
- ALUOp=01

PCWrite

PCSource=10

ALUSrcA=1
- ALUSrcB=10
- ALUOp=00

ALUSrcA=1
- ALUSrcB=10
- ALUOp=00

ALUSrcA=1
- ALUSrcB=00
- ALUOp=01

MemWrite
- IorD=1

Addi
- ALUSrcA=1
- ALUSrcB=00
- ALUOp=10

RegDst=0
- MemToReg=0
- RegWrite=1

I-type completion

RegDst=0
- MemToReg=0
- RegWrite=1

I-type completion

RegDst=1
- MemToReg=0
- RegWrite=1

R-type completion

RegWrite
- MemToReg=1
- RegDst=0

Op==lw

Op==sw

Op==beq

Jump execution

Jump execution

Branch completion

Branch completion
Flow of instructions

- **Normal/expected flow**
  - Sequential: common code operating on data
  - Non-sequential: branches and jumps

- **Unexpected flow**
  - Internal (*Exception/Trap*)
    - Arithmetic overflow
    - Undefined instruction
    - Unauthorized access to memory
    - Requesting service from operating system (system call)
    - Hardware failure
  - External (*Interrupt*)
    - Request for “attention” from an I/O device
    - Hardware failure
Supporting exceptions and interrupts

- **Hardware support (minimum necessary)**
  - Stop executing an instruction
    - Maintain valid processor and computation state
  - Allow to identify cause
    - Flag bits in a special register
    - Identifier of exception type
  - Store address of instruction that caused exception
    - Allows re-executing or skipping an instruction on resume
  - Jump to exception/interrupt handler
    - Single address for all exceptions/interrupts
    - Multiple addresses corresponding to exception type
Arithmetic overflow exception

R-type execution

From 1

ALUSrcA=1
ALUSrcB=00
ALUOp=10

To 0

IntCause=1
CauseWrite
ALUSrcA=0
ALUSrcB=01
ALUOp=01
EPCWrite
PCWrite
PCSource=11

R-type completion

RegDst=1
MemToReg=0
RegWrite=1

Overflow
Undefined instruction exception

From 1

Undefined instruction

14

To 0

Undefined instruction

ALUSrcA=0
ALUSrcB=01
ALUOp=01
IntCause=0
CauseWrite
PCSource=11
EPCWrite
PCWrite
Multi-cycle datapath control (4)
Supporting exceptions and interrupts (2)

- **Software handler**
  - Store the current state of computation
    - Save contents of CPU registers to memory
  - Determine the cause of exception/interrupt and execute the corresponding handler routine
    - Deal with I/O device
    - Deal with memory management
    - Continue/terminate current process
    - Switch to another process
  - Restore state of current (next) process
  - Resume execution (jump into) of current (next) process
    - Restart instruction that caused an exception
    - Continue from next instruction
Multi-cycle datapath performance

- **Instruction mix**
  - 30% load (5ns), 10% store (5ns)
  - 50% add (4ns), 10% mul (20ns)

- **Single-cycle datapath (clock period 20ns, CPI = 1)**
  - 20ns per instruction → 50 MIPS

- **Coarse-grained multi-cycle datapath (clock period 5ns)**
  - \( CPI \approx (90\% \times 1) + (10\% \times 4) = 1.3 \)
  - 6.5ns per instruction → 153 MIPS

- **Fine-grained multi-cycle datapath (clock period 1ns)**
  - \( CPI \approx (30\% \times 5) + (10\% \times 5) + (50\% \times 4) + (10\% \times 20) = 6 \)
  - 6ns per instruction → 166 MIPS
Implementing a sequential controller

- Implementing a finite-state automaton
  - State + transition conditions = memory + combinational logic → sequential logic
    - Implementation depends on internal state representation
  - Sequential circuitry
    - 1 flip-flop per state (only one active at a time), active state shifted through enabling gates between flip-flops
    - State register + combinational logic
  - Simple sequencer + control memory
    - Micro- and nano- programming
Implementing a sequential controller

- **State register**
- **Control logic**
  - Combinational logic
  - ROM, FPGA

![Diagram of a sequential controller with inputs, outputs, state register, instruction register, opcode field, NS0, NS1, NS2, NS3, ALUOp, MemRead, PCWrite, and combinational logic connections.](image-url)
Next state is next control ROM address
Control memory address select logic
Horizontal micro-instructions

- **Direct representation of control signals**
  - Control memory contains raw control signals
  - Micro-instruction = set of control signal values
    - No need to decode (fast)
    - Any combination is possible (flexible)
    - Requires a lot of space
Vertical micro-instructions

- **Encoded representation of control signals**
  - Microinstructions identify valid combinations of control signals
    - Decoded intro actual control signals using a decoder
    - Reduces space at the cost of flexibility and latency
Nano-programming

- Combines horizontal & vertical encoding
  - Microprogram memory only contains numbers representing valid combinations of control signals (vertical format)
  - Decoding to horizontal format is realized using another memory (instead of a combinational circuit) which contains the control signal combination corresponding to microprogram code
  - Significantly reduces the amount of space required to store the microprogram, but increases decoding latency
Micro- vs nano-programming

Total Area = \( n \times w = 2048 \times 41 = 83,968 \) bits

Microprogram Area = \( n \times k = 2048 \times 7 = 14,336 \) bits
Nanoprogram Area = \( m \times w = 100 \times 41 = 4100 \) bits
Total Area = 14,336 + 4100 = 18,436 bits