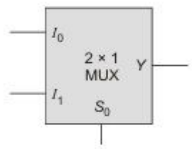
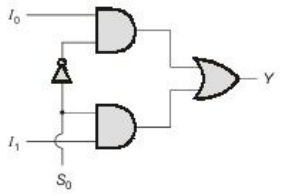
Section A

2 : 1 Multiplexer (2 × 1 MUX)

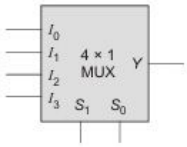


S_0	Y
0	I_0
1	I_1

$Y = \bar{S}_0 I_0 + S_0 I_1$

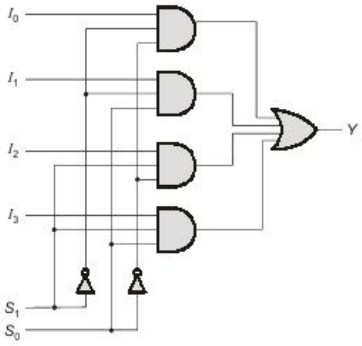


4 : 1 Multiplexer (4 × 1 MUX)

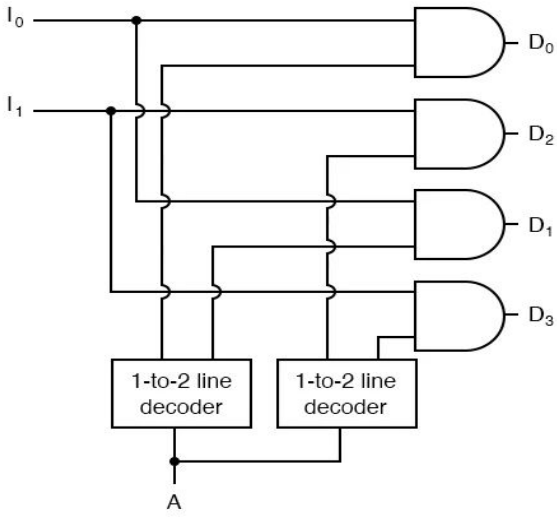


S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

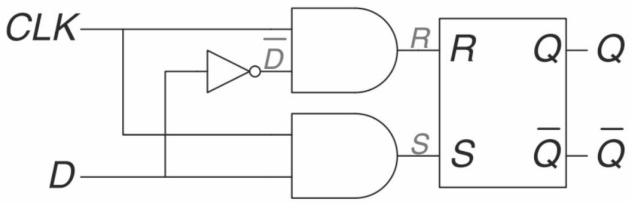
$Y = \bar{S}_1\bar{S}_0 I_0 + \bar{S}_1 S_0 I_1 + S_1\bar{S}_0 I_2 + S_1 S_0 I_3$



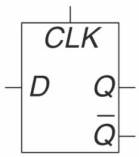
DMux



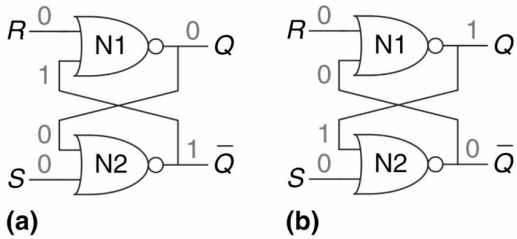
D-Latch



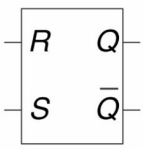
CLK	D	$\bar{D}$	S	R	Q	$\bar{Q}$
0	X	$\bar{X}$	0	0	Q_{prev}	$\bar{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0



SR(RS)-Latch



Case	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prev}	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0



D Latch:

- **Advantage:** No invalid state; input directly controls output when enabled.
- **Disadvantage:** Output is transparent (changes with input D as long as enable is active).

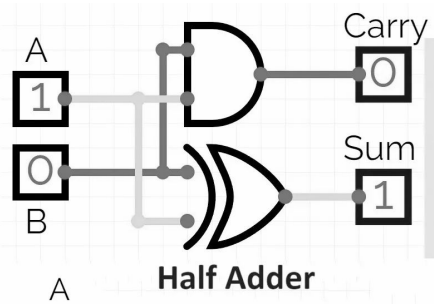
SR (RS) Latch:

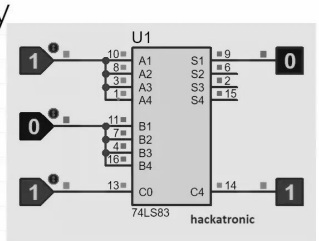
- **Advantage:** Simplest memory element.
- **Disadvantage:** Invalid state (S=R=1 is forbidden/unpredictable).

SR Latch vs. D-Flip-Flop (DFF):

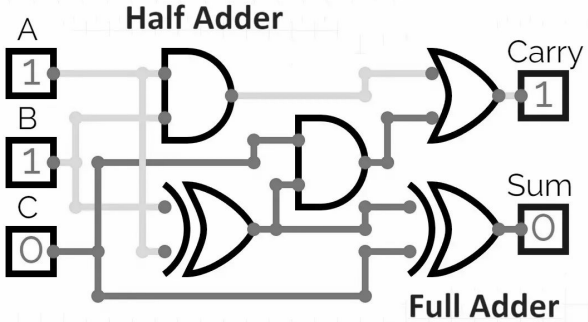
Triggering: DFFs use clock **edges**; latches use clock **levels**.

DFF Construction: Can be made with AND/NAND gates, behavior remains consistent.

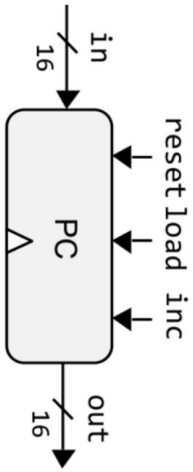




A	B	Sum (S)	Carry (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

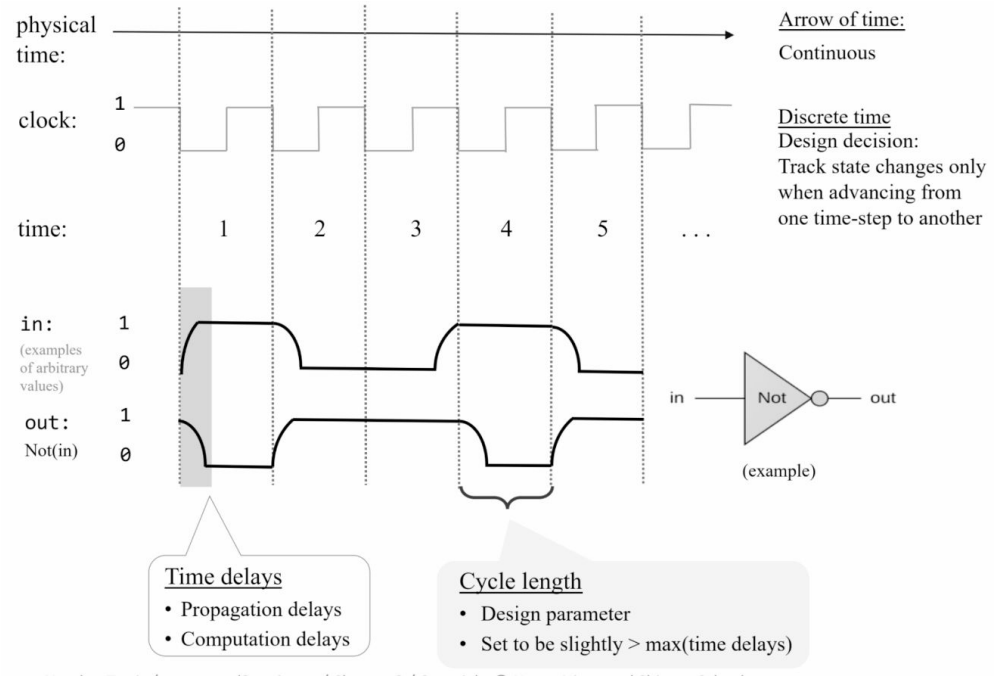


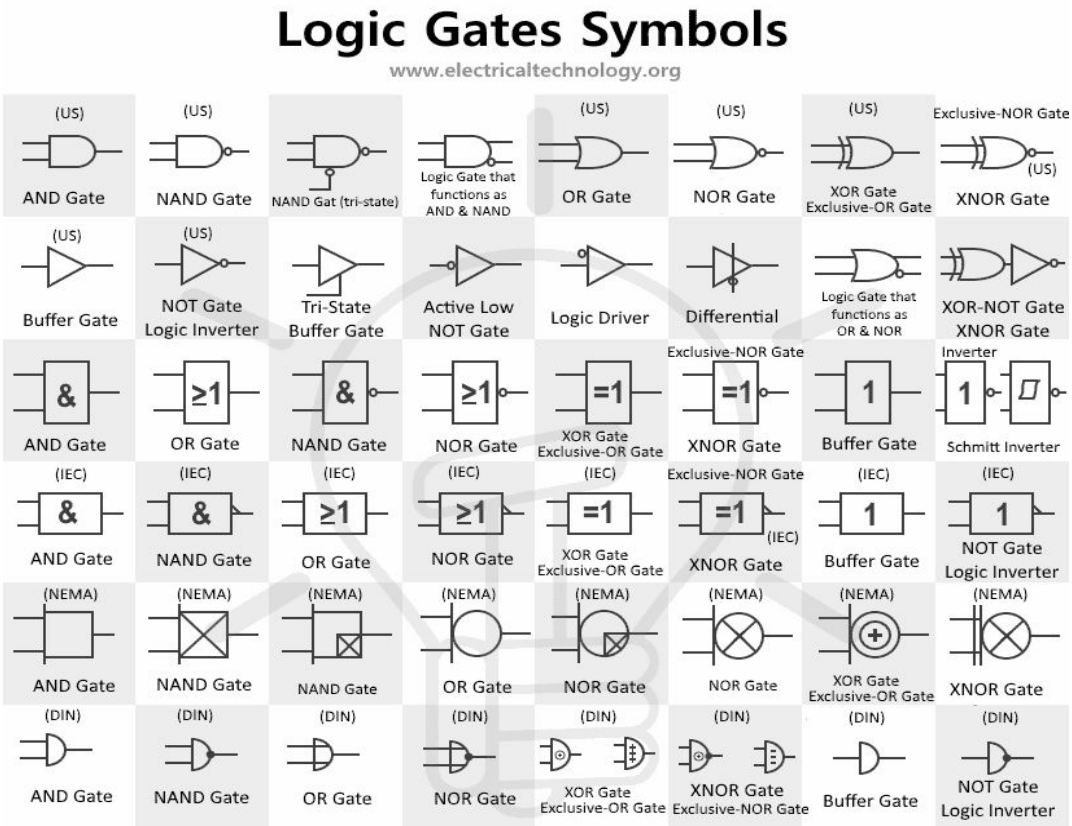
A	B	Cin	Sum (S)	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



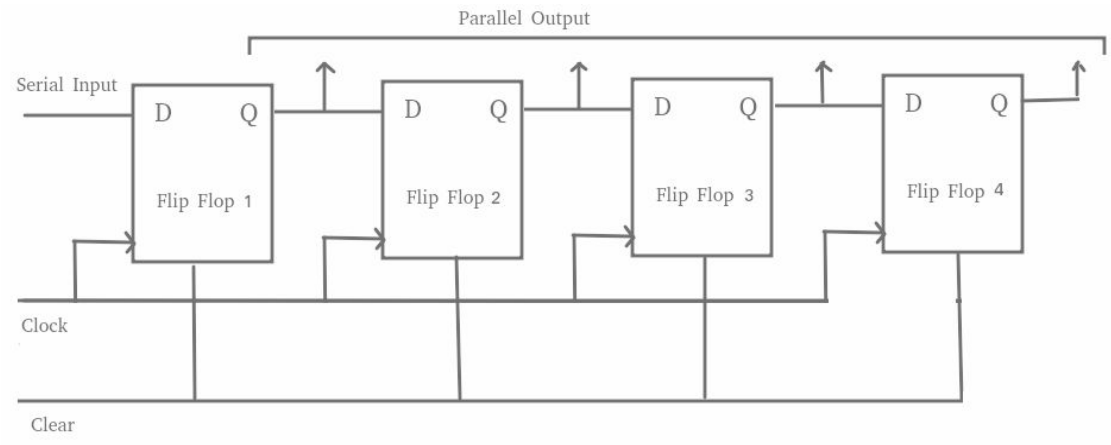
Program counter (PC)

```
if      reset(t): out(t+1) = 0
else if load(t): out(t+1) = in(t)
else if inc(t): out(t+1) = out(t) + 1
else    out(t+1) = out(t)
```





Shift-register



Logic Circuit Types

Combinational / Sequential Circuits

- **Comb. Function:** Produce output directly from current inputs.
- **Memory:** None; output solely depends on present input.

- **Seq. Function:** Store information (have memory). Output can be fed back as input.
- **Dependency:** Output depends on current inputs and stored state (past outputs).
- **Timing Specs:** Critical for correct operation.
    - **Setup Time:** Input must be stable *before* clock edge.
    - **Hold Time:** Input must be stable *after* clock edge.
- **Loop Handling:** Insert **registers** to break combinational loops and synchronize feedback with the clock.

Boolean Expression Forms

- **Sum-of-Products (SOP):** ORing of ANDed terms (e.g., `AB + C`).
- **Product-of-Sums (POS):** ANDing of ORed terms (e.g., `(A+B)(C)`).
- **Minterm Form (Canonical SOP):** Sum of product terms where each term includes **all variables** (true or complemented). Represents when the function is **TRUE**.
- **Maxterm Form (Canonical POS):** Product of sum terms where each term includes **all variables** (true or complemented). Represents when the function is **FALSE**.

Boolean Algebra & De Morgan's Theorem

- **Common Boolean Symbols:**
    - AND: `·` or `∧` (or no symbol, e.g., `AB`)
    - OR: `+` or `∨`
    - NOT: `'` (prime), `¬`, or an overbar (e.g., `A'`)
- **De Morgan's Theorem:**
    - **(A · B)' = A' + B'** (NOT (A AND B) = (NOT A) OR (NOT B))
        - Complement of a product is the sum of complements.
    - **(A + B)' = A' · B'** (NOT (A OR B) = (NOT A) AND (NOT B))
        - Complement of a sum is the product of complements.
    - **Utility:** Allows moving inversion bubbles around logic gates to simplify circuits.

Common Combinational Circuits

- **Multiplexer (MUX):** Selects one of several input signals to pass to a single output.
    - Example: `Y = S · A + S' · B` (S is the select line)
- **Decoder:** Converts `n` input lines to `2^n` unique output lines.

Circuit Optimization ⚙️

- **Minimal Expressions:** Lead to **reduced hardware complexity** and **lower cost**.

Synchronous Design & Timing

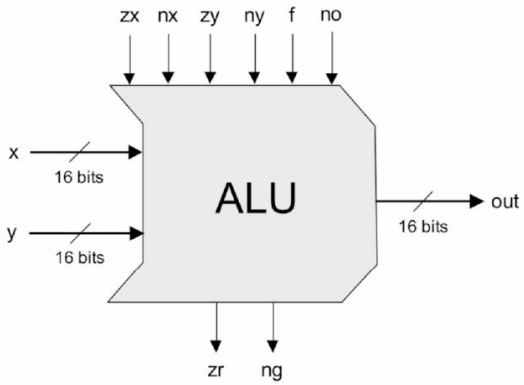
- **Synchronous Circuits:** Use a single, common **clock** for all registers.
- **Registers:** Store multiple bits; use a common clock.
- **Stable Circuits:** Maintain stable states (Note: research more for deeper understanding).
- **Clock Skew:** Timing differences in clock signal arrival at different parts of the circuit.
- **Pipelining:** Inserting registers to manage circuit timing and improve throughput.
- **Synchronizer:** Aligns asynchronous inputs with the system clock.

Hardware Design & Arithmetic

- **Hardware Description Language (HDL) (e.g., Nand2Tetris HDL):** Used to simulate and test hardware logic.
- **Adders:**
    - **Ripple-Carry Adder:** Slow due to **carry propagation delay** (carry must ripple through all bits).
    - **Carry-Lookahead Adder:** Faster; **predicts carries** in parallel.
- **Comparators:** Determine if one value is greater than, less than, or equal to another.
- **Multiplication:** Implemented using **AND gates** (for partial products) and **adders**. Slower than adders due to multiple additions.
- **Division:** Implemented using **subtractors**.
- **Counters:** Increment a value on clock edges.
    - Example: A 4-bit ripple counter counts from 0 to 15 (2^4-1).
- **Memory Arrays:** Built using **flip-flops** or **transistors** in grid structures.
- **Programmable Logic Array (PLA):**
    - **Benefits:** Programmability and regular structure.
- **Fixed-Point Arithmetic Units:** Used for calculations involving **fractional numbers**.
-
- **ALU** - always for A and L operations - dont let it trick you
- The HACK **databus** is **16 bits wide**
- D = general purpose, A = AR, M = Memory[A]
- HACK memory can hold 32K instructions

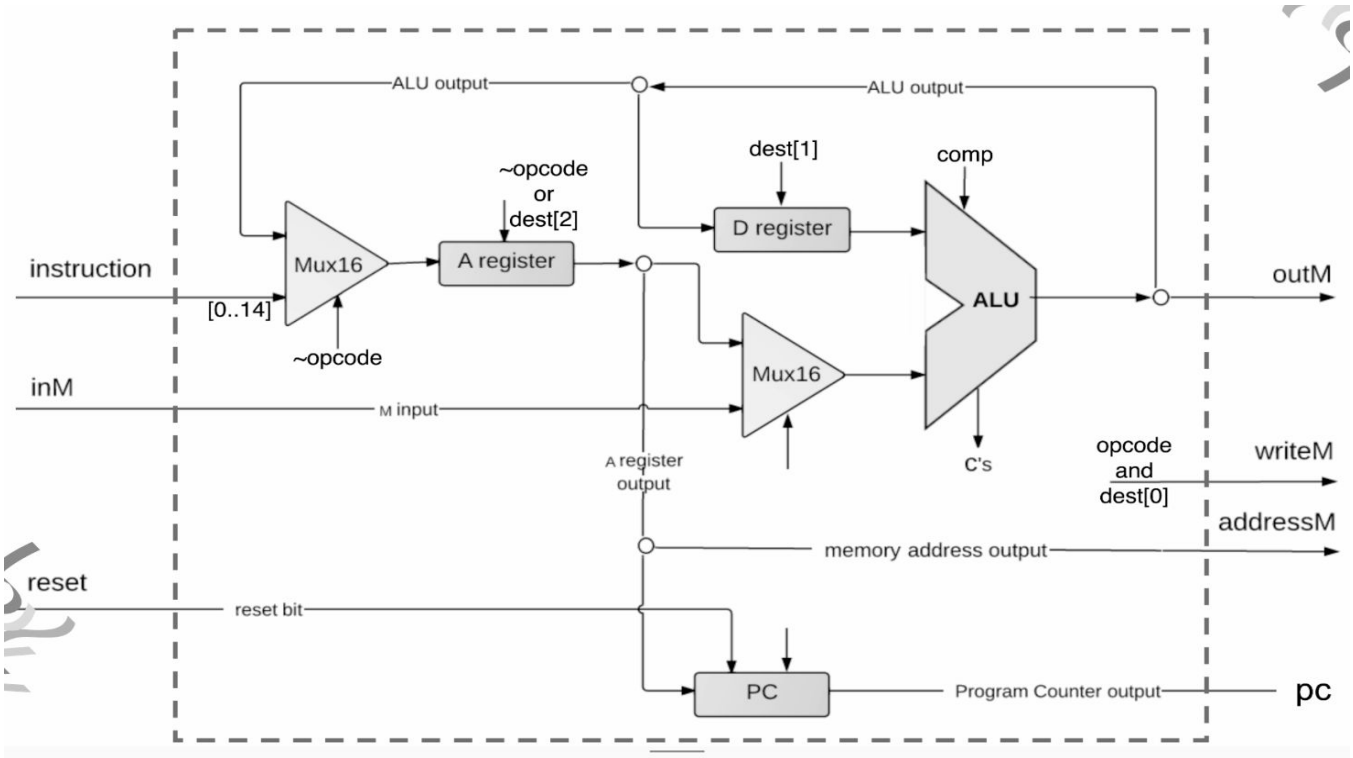
Feature	RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
Emphasis	Software	Hardware
Control Unit	Hardwired only	Hardwired or microprogrammed
Transistor Use	More registers	Storing complex instructions
Instruction Size	Fixed	Variable
Arithmetic Ops	Register to Register only	Register to Register, Register to Memory, Memory to Memory
Number of Registers	More	Less
Code Size	Large	Small
Cycles/Instruction	Single clock cycle	Multiple clock cycles
Instruction Word	Fits in one word	Can be larger than one word
Addressing Modes	Simple & limited	Complex & numerous
Full Name	Reduced Instruction Set Computer	Complex Instruction Set Computer
# of Instructions	Fewer	More
Power Consumption	Low	High
Pipelining	Highly pipelined	Less pipelined
RAM Requirement	More	Less

Appendix 1. Hack processor ALU



pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Section B:



Appendix 2. Hack language commands encoding

A instruction

Symbolic: `@`*xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: `0` *vvvvvvvvvvvvvvv* (*vv ... v* = 15-bit value of *xxx*)

C instruction

Symbolic: *dest* = *comp*; *jump* (*comp* is mandatory. If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted)

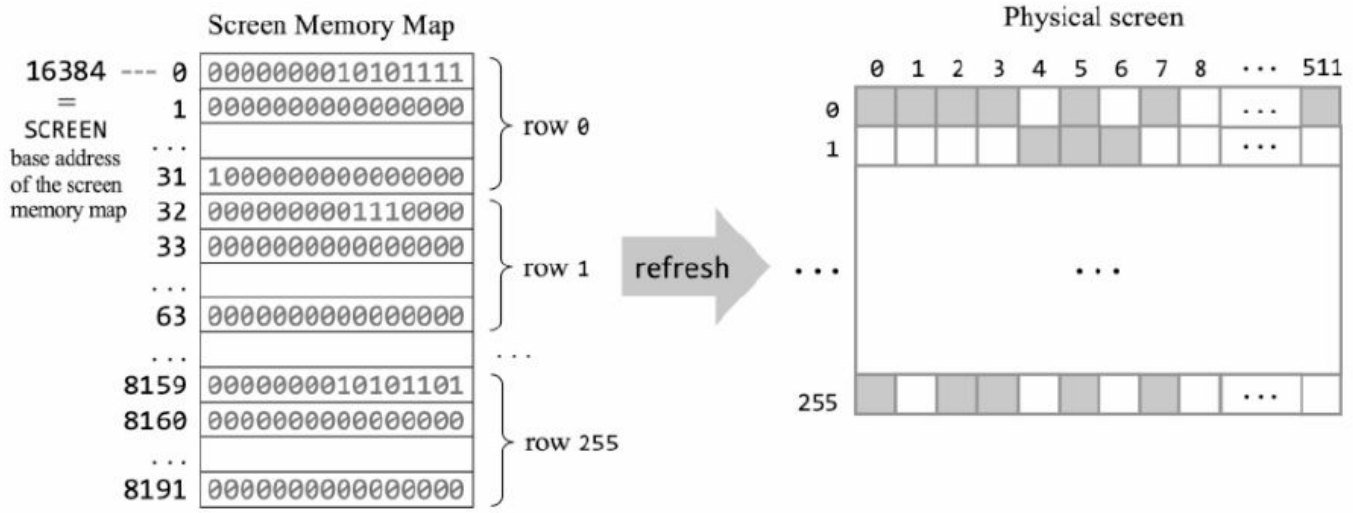
Binary: `111`*accccccdddjjj*

comp		*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
		a == 0			*a* == 1		

dest	*d*	*d*	*d*	Effect: store *comp* in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register (reg)
DM	0	1	1	RAM[A] and D reg
A	1	0	0	A reg
AM	1	0	1	A reg and RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A reg, D reg, and RAM[A]

jump	*j*	*j*	*j*	Effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	unconditional jump

Appendix 3. Hack computer screen memory mapping



Computation:
0 Zero
1 One
- 1 Negative one

Unary:
register
! register Not
- register Negate

Binary:
register + 1 Add one
register - 1 Subtract one
register op register
+ Add
- Subtract
* Multiply
/ Divide
& And
| Or
^ Xor

<< Logical shift left
>> Logical shift right

Note: The registers should be different (ex. D + D is not supported)

address	variableName(s)
0	R0, SP
1	R1, LCL
2	R2, ARG
3	R3, THIS
4	R4, THAT
5	R5
6	R6
...	...
15	R15
*	SCREEN
*	KBD
*	MOUSE

Loading an Address (Symbol)

@SCREEN // A = 16384
@KBD // A = 24576
@myLabel // A = address of (myLabel)

KBD	24576 (0x6000)	Keyboard memory-mapped register
SCREEN	16384 (0x4000)	Start of video memory (RAM[16384]–RAM[24575])

SP Stack pointer
LCL Base of local segment
ARG Base of argument segment
THIS Base of "this" segment (e.g., object fields)
THAT Base of "that" segment (used like THIS)
R5-R12 general purpose temporary variables
R12-R15 general purpose

dest	Stores To
null	Result discarded
M	Memory[A]
D	D Register
MD	M and D
A	A Register
AM	A and M
AD	A and D
AMD	A, M, and D

The HACK databus is 16 bits wide

HACK memory can hold 32K instructions

3. **L-Instruction (Label Pseudo-Instruction):** Declares a label (symbol) that marks a specific ROM address.
   - **Syntax:** `(LABEL_NAME)`
   - **Example:**

```
(LOOP)  // Declares a label named LOOP, marking the address of the next ins
```

RISC-V

-The stack is an area of memory set aside for use by functions and local variables

-The **stack pointer (sp)** points to the bottom of the stack, which grows **downwards** to lower addresses We allocate memory on the stack by decrementing the stack pointer **sp**.

-We can then save registers onto the stack using the **sw** (store word) instruction

Feature	Multiplexer (Mux)	Demultiplexer (Demux)	Encoder	Decoder
Primary Goal	Data Selection	Data Distribution	Data Compression/Representation	Data Interpretation/Selection
Data Lines	Multiple Inputs → Single Output	Single Input → Multiple Outputs	Multiple Inputs → Fewer Outputs (coded)	Fewer Inputs (coded) → Multiple Outputs
Control	Select Lines	Select Lines	Inputs themselves determine output code	Input code determines active output
Output Nature	Passes through one selected input data	Passes input data to one selected output	Binary code representing active input	Activates one specific output line
No. of Outputs	1 data output	2^S data outputs (where S is #select lines)	Typically $\log_2(N)$ outputs (for N inputs)	Typically 2^N outputs (for N input lines)
Use Case	Selecting between different data sources	Sending data to one of many destinations	Converting parallel inputs to binary form (e.g., keyboard)	Activating a specific device/memory location/operation based on a binary code (e.g., instruction decoding)

RISC-V function example:

```
fun_one:
   addi sp, sp, -16  # allocate 16 bytes on
stack
   sw   ra, 12(sp)   # store return address
on stack

   # do some fun stuff

   call fun_two  # call another function

   # do some more fun stuff

   lw   ra, 12(sp)  # load return address
from stack
   addi sp, sp, 16  # restore stack pointer

   ret  # return from fun_one
```

```
#
# Risc-V Assembler program to print "Hello World!"
# to stdout.
#
# a0-a2 - parameters to linux function services
# a7 - linux function number
#

.global _start      # Provide program starting address to linker

# Setup the parameters to print hello world
# and then call Linux to do it.

_start: addi  a0, x0, 1      # 1 = StdOut
        la    a1, helloworld # load address of helloworld
        addi  a2, x0, 13     # length of our string
        addi  a7, x0, 64     # linux write system call
        ecall                # Call linux to output the string

# Setup the parameters to exit the program
# and then call Linux to do it.

        addi    a0, x0, 0   # Use 0 return code
        addi    a7, x0, 93  # Service command code 93 terminates
        ecall               # Call linux to terminate the program

.data
helloworld:      .ascii "Hello World!\n"
```

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Microarchitecture is the Internal organization of datapath and control for a CPU

Finite State Machines dictate the datapaths in a CPU

Pipeline hazard = Conflict from instruction overlap

ALU - always for A and L operations - dont let it trick you

TLB stores mappings from virtual memory to physical memory

Risc-V Instructions

Arithmetic Instructions

- **add rd, rs1, rs2**: Adds the values in registers rs1 and rs2, stores the result in register rd. (Similar to MIPS add)
- **sub rd, rs1, rs2**: Subtracts the value in register rs2 from rs1, stores the result in rd. (Similar to MIPS sub)
- **addi rd, rs1, imm**: Adds the immediate value imm to the value in register rs1, stores the result in rd. (Similar to MIPS addi)

Logical Instructions

- **and rd, rs1, rs2**: Performs a bitwise AND of values in rs1 and rs2, stores the result in rd. (Similar to MIPS and)
- **or rd, rs1, rs2**: Performs a bitwise OR of values in rs1 and rs2, stores the result in rd. (Similar to MIPS or)
- **xor rd, rs1, rs2**: Performs a bitwise XOR of values in rs1 and rs2, stores the result in rd.
- **andi rd, rs1, imm**: Performs a bitwise AND of the value in rs1 with the immediate imm, stores the result in rd.
- **ori rd, rs1, imm**: Performs a bitwise OR of the value in rs1 with the immediate imm, stores the result in rd.
- **xori rd, rs1, imm**: Performs a bitwise XOR of the value in rs1 with the immediate imm, stores the result in rd.

Data Transfer Instructions

- **lw rd, offset(rs1)**: Loads a word from the memory address calculated by rs1 + offset into register rd. (Similar to MIPS lw)
- **sw rs2, offset(rs1)**: Stores the word from register rs2 to the memory address calculated by rs1 + offset. (Similar to MIPS sw)

Control Flow Instructions

- **beq rs1, rs2, label**: Branches to label if the values in rs1 and rs2 are equal. (Similar to MIPS beq)
- **bne rs1, rs2, label**: Branches to label if the values in rs1 and rs2 are not equal.
- **blt rs1, rs2, label**: Branches to label if the value in rs1 is less than the value in rs2 (signed).
- **bge rs1, rs2, label**: Branches to label if the value in rs1 is greater than or equal to the value in rs2 (signed).
- **jal rd, label**: Jumps to label and stores the address of the next instruction (PC+4) in register rd (often ra for return address). (Similar to MIPS jal)
- **jalr rd, rs1, offset**: Jumps to the address rs1 + offset and stores PC+4 in rd. (Used for returning from functions when rs1 is ra and offset is 0, and rd is x0 or zero)

Set Less Than Instructions

- **slt rd, rs1, rs2**: Sets rd to 1 if rs1 is less than rs2 (signed), otherwise 0. (Similar to MIPS slt)
- **slti rd, rs1, imm**: Sets rd to 1 if rs1 is less than the immediate imm (signed), otherwise 0.
- **sltu rd, rs1, rs2**: Sets rd to 1 if rs1 is less than rs2 (unsigned), otherwise 0.
- **sltiu rd, rs1, imm**: Sets rd to 1 if rs1 is less than the immediate imm (unsigned), otherwise 0.

Hack assembler

- **Role**: Translates Hack assembly language (.asm source code file) into 16-bit binary machine code (.hack).
- **Process**:
  1. **First Pass**: Builds the **Symbol Table**.
     - Scans for label declarations (LABEL) and assigns them the address of the next instruction in ROM (instruction memory address). L-instructions ((LABEL)) are for jumps.
     - Predefined symbols: R0-R15 (0-15), SCREEN (16384), KBD (24576), SP(0), LCL(1), ARG(2), THIS(3), THAT(4).
     - Variable symbols (e.g., @i, @sum): Assigned RAM addresses starting from 16 by the assembler.
  2. **Second Pass**: Translates instructions to binary.
     - Replaces symbols (labels and variables) with their numeric addresses from the symbol table.
     - Converts A-instructions and C-instructions to their binary format.
- **Assembler Architecture Modules**:
  - **Parser**: Reads assembly instructions, parses them, identifies command type (A, C, L), and provides access to their components (symbol, dest, comp, jump).
  - **Code**: Translates symbolic mnemonics of C-instructions (comp, dest, jump) into their binary codes.
  - **SymbolTable**: Manages symbols and their corresponding numeric addresses (addEntry, contains, getAddress).

MIPS assembly instructions:

Arithmetic Instructions

- **add rd, rs, rt**: Adds the values in registers rs and rt, stores the result in register rd. (R-type)
  - Example: add $s0, $s1, $s2
- **sub rd, rs, rt**: Subtracts the value in register rt from rs, stores the result in rd. (R-type)=
  - Example: sub $t0, $t3, $t5
- **addi rt, rs, imm**: Adds the sign-extended immediate value imm to the value in register rs, stores the result in rt. (I-type)
  - Example: addi $s0, $s1, 5

Logical Instructions

- **and rd, rs, rt**: Performs a bitwise AND of values in rs and rt, stores the result in rd. (R-type)
- **or rd, rs, rt**: Performs a bitwise OR of values in rs and rt, stores the result in rd. (R-type)
- **slt rd, rs, rt**: Sets rd to 1 if rs is less than rt (signed), otherwise 0. (R-type)

Data Transfer Instructions

- **lw rt, offset(rs)**: Loads a word from the memory address calculated by rs + sign-extended offset into register rt. (I-type)
  - Example: lw $t2, 32($s0)
- **sw rt, offset(rs)**: Stores the word from register rt to the memory address calculated by rs + sign-extended offset. (I-type)
  - Example: sw $s1, 4($t1)

Control Flow Instructions

- **beq rs, rt, label**: Branches to label if the values in rs and rt are equal. The target address is PC-relative. (I-type, branch format)
- **j label**: Jumps unconditionally to label. The target address is formed using pseudo-direct addressing. (J-type)
- **jal label**: Jumps and links. Jumps unconditionally to label and stores the address of the next instruction (PC+4) in the return address register $ra ($31). Used for procedure calls. (J-type)
  - Example: jal sum

Section C:

Combinational Circuits

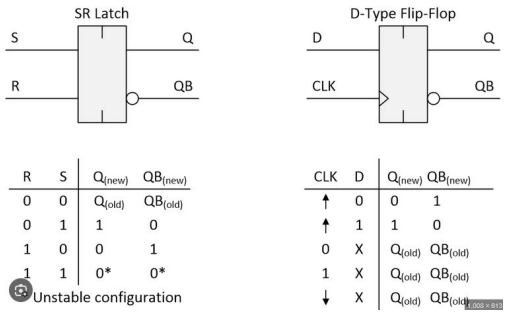
- **Function:** Produce output directly from current inputs.
- **Memory:** None; output solely depends on present input.

Sequential Circuits

- **Function:** Store information (have memory). Output can be fed back as input.
- **Dependency:** Output depends on current inputs and stored state (past outputs).
- **Timing Specs:** Critical for correct operation.
    - **Setup Time:** Input must be stable *before* clock edge.
    - **Hold Time:** Input must be stable *after* clock edge.
- **Loop Handling:** Insert **registers** to break combinational loops and synchronize feedback with the clock.

Latches vs. Flip-Flops

- **SR Latch vs. D-Flip-Flop (DFF):**
    - **Triggering:** DFFs use clock **edges**; latches use clock **levels**.
    - **DFF Construction:** Can be made with AND/NAND gates, behavior remains consistent.
    -



R	S	$Q_{(new)}$	$QB_{(new)}$
0	0	$Q_{(old)}$	$QB_{(old)}$
0	1	1	0
1	0	0	1
1	1	0*	0*

Unstable configuration

CLK	D	$Q_{(new)}$	$QB_{(new)}$
↑	0	0	1
↑	1	1	0
0	X	$Q_{(old)}$	$QB_{(old)}$
1	X	$Q_{(old)}$	$QB_{(old)}$
↓	X	$Q_{(old)}$	$QB_{(old)}$

Synchronous Design & Timing

- **Synchronous Circuits:** Use a single, common **clock** for all registers.
- **Registers:** Store multiple bits; use a common clock.
- **Stable Circuits:** Maintain stable states (Note: research more for deeper understanding).
- **Clock Skew:** Timing differences in clock signal arrival at different parts of the circuit.
- **Pipelining:** Inserting registers to manage circuit timing and improve throughput.
- **Synchronizer:** Aligns asynchronous inputs with the system clock.

Hardware Design & Arithmetic

- **Hardware Description Language (HDL) (e.g., Nand2Tetris HDL):** Used to simulate and test hardware logic.
- **Adders:**
    - **Ripple-Carry Adder:** Slow due to **carry propagation delay** (carry must ripple through all bits).
    - **Carry-Lookahead Adder:** Faster; **predicts carries** in parallel.
- **Comparators:** Determine if one value is greater than, less than, or equal to another.
- **Multiplication:** Implemented using **AND gates** (for partial products) and **adders**. Slower than adders due to multiple additions.
- **Division:** Implemented using **subtractors**.
- **Counters:** Increment a value on clock edges.
    - Example: A 4-bit ripple counter counts from 0 to 15 (2^4-1).
- **Memory Arrays:** Built using **flip-flops** or **transistors** in grid structures.
- **Programmable Logic Array (PLA):**
    - **Benefits:** Programmability and regular structure.
- **Fixed-Point Arithmetic Units:** Used for calculations involving **fractional numbers**.

Single-Cycle Processor

- **Concept**: Each instruction executes in a single, long clock cycle.
- **Problems**:
  - Clock cycle time is determined by the slowest instruction.
  - Requires multiple, dedicated hardware units (e.g., adders for PC increment and ALU) that might not always be used.
  - Typically uses separate instruction and data memories (Harvard architecture aspect).

MIPS Architecture Basics

- **Type**: RISC (Reduced Instruction Set Computer) architecture. Key principle: Simplicity favors regularity.
- **Registers**:
  - 32 general-purpose 32-bit registers ($s0-$s7, $t0-$t9, etc.).
  - Register $0 ($zero) always holds the value 0.
- **Instruction Formats** (all 32 bits long):
  - **R-type (Register)**: Operates on three registers (e.g., add $rd, $rs, $rt).
    - Fields: opcode (6) | rs (5) | rt (5) | rd (5) | shamt (5) | funct (6).
  - **I-type (Immediate)**: Operates on two registers and a 16-bit immediate value (e.g., lw $rt, offset($rs), addi $rt, $rs, imm).
    - Fields: opcode (6) | rs (5) | rt (5) | immediate (16).
  - **J-type (Jump)**: Operates on a 26-bit address (e.g., j target).
    - Fields: opcode (6) | address (26). jal (jump and link) is used for procedure calls.
- **Addressing Modes**:
  - **Register-Only**: Operands are in registers (R-type).
  - **Immediate**: One operand is a constant within the instruction (e.g., addi).
  - **Base Addressing**: Address is register_value + sign-extended_offset (e.g., lw, sw).
    - MIPS is byte-addressable, but words are 4 bytes. lw $t0, 4($sp) loads word from address $sp + 4 into $t0.
  - **PC-Relative Addressing**: Branch address is PC + 4 + (sign-extended_offset << 2) (e.g., beq, bne). beq branches if registers are equal.
  - **Pseudo-Direct Addressing**: Jump address is formed by (PC+4)[31:28] | address_from_instruction << 2 (J-type).

MIPS Datapath Elements

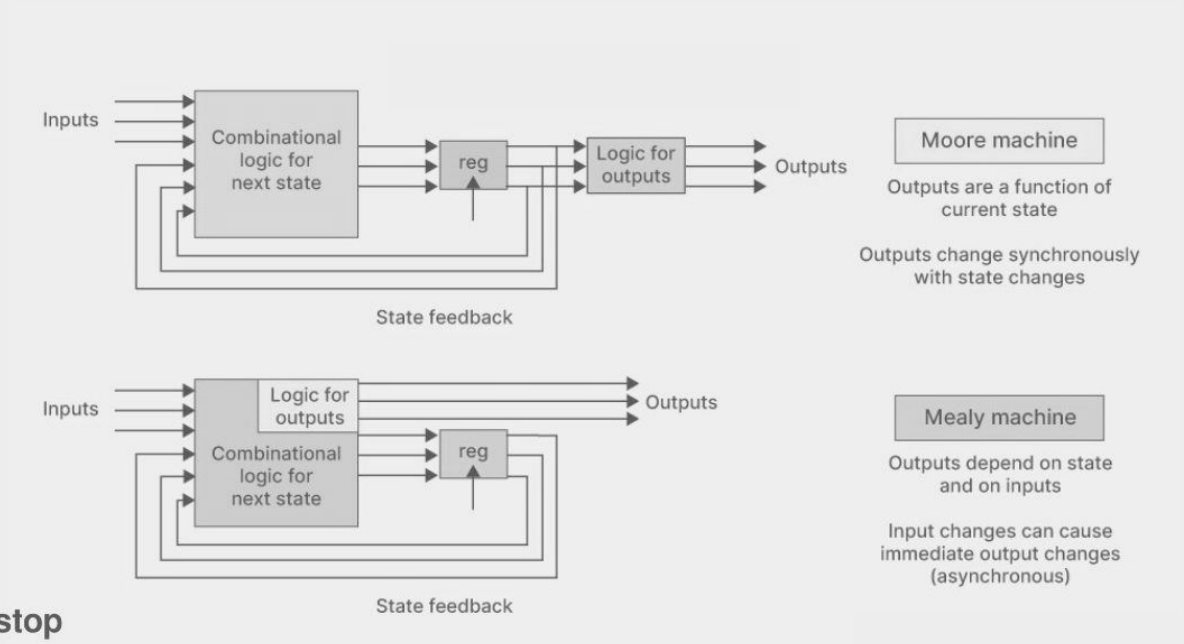
- **Program Counter (PC)**: Holds address of current/next instruction.
- **Instruction Memory**: Stores instructions.
- **Register File**: Stores general-purpose registers. Has read and write ports.
- **ALU (Arithmetic Logic Unit)**: Performs calculations.
- **Data Memory**: Stores data. (Unified with Instruction Memory in multicycle).
- **Sign Extender**: Extends 16-bit immediates to 32 bits.

Multi-Cycle Processor

- **Motivation**: Overcome single-cycle limitations by breaking instruction execution into multiple shorter clock cycles. Allows different instructions to take different numbers of cycles. Uses unified instruction/data memory.
- **Introduces intermediate registers**: IR (Instruction Register), MDR (Memory Data Register), A, B (ALU input registers), ALUOut.
- **Instruction Execution Steps (LW example)**:
  1. **Fetch (IF)**: IR = Mem[PC], PC = PC + 4. (1 cycle)
  2. **Decode (ID)**: Decode IR. Read registers A = RegFile[rs], B = RegFile[rt]. ALUOut = PC + (sign_extend(IR[15:0]) << 2) (for branches, target address computation). (1 cycle)
  3. **Execute (EX)**:
     - Memory Ref (lw, sw): ALUOut = A + sign_extend(IR[15:0]) (address calculation). (1 cycle)
     - R-type: ALUOut = A op B. (1 cycle)
     - Branch (beq): If (A==B) then PC = ALUOut (ALUOut was branch target from ID). (1 cycle)
     - Jump (j): PC = (PC[31:28]) | (IR[25:0] << 2) (this is done in ID/EX for J-type). (1 cycle)
  4. **Memory (MEM)**:
     - LW: MDR = Mem[ALUOut]. (1 cycle)
     - SW: Mem[ALUOut] = B. (1 cycle)
  5. **Write Back (WB)**:
     - LW: RegFile[IR[20:16]] = MDR. (1 cycle)
     - R-type: RegFile[IR[15:11]] = ALUOut. (1 cycle)
- **Control Unit**: Implemented as a Finite State Machine (FSM). Each state corresponds to one clock cycle of an instruction's execution.
- **Cycles per Instruction (CPI)** for MIPS multi-cycle (example from slides/H&H):
  - LW: 5 cycles
  - SW: 4 cycles
  - R-type: 4 cycles
  - I-type (like addi): 4 cycles
  - BEQ: 3 cycles
  - J: 3 cycles

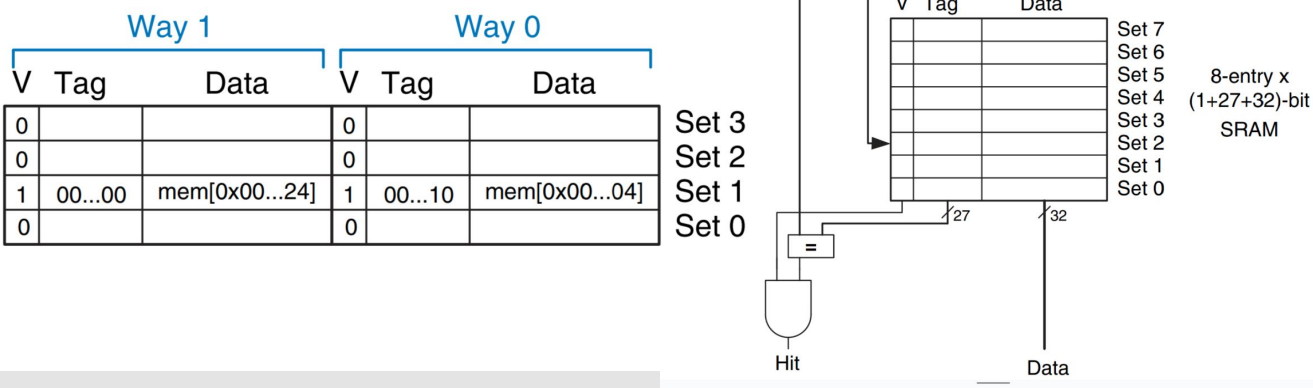
Feature	Memory Management Unit (MMU)	Translation Lookaside Buffer (TLB)
Nature	A complete hardware unit for memory management.	A specialized cache *within* the MMU.
Primary Role	Translates virtual to physical addresses, enforces protection, manages virtual memory.	Speeds up address translation by caching recent translations.
Scope	Broad responsibilities for memory management.	Specific to caching address translations.
Dependency	Operates independently, though often works with the OS.	Relies on the MMU to perform full page walks on a cache miss.
Performance Impact	Enables virtual memory and protection, but can be slow without a TLB.	Significantly reduces latency of address translation, improving overall system speed.

Mealy vs Moore Machine
What's the difference?



unstop

1. **Direct Mapping:**
    - Each main memory block can only be placed in one specific cache line.
    - It's simple to implement, as the main memory address directly dictates the cache location.
    - **Disadvantage:** Can lead to "conflict misses" if frequently accessed blocks map to the same cache line, even if the cache isn't full.
2. **Fully Associative Mapping:**
    - Any main memory block can be placed in *any* available cache line.
    - Offers the most flexibility and generally achieves the highest hit rate, as it minimizes conflict misses.
    - **Disadvantage:** Requires a complex and expensive comparison circuit (Content-Addressable Memory or CAM) to check all cache tags simultaneously, making it less practical for large caches.
3. **Set-Associative Mapping:**
    - A hybrid approach that balances the trade-offs of direct and fully associative mapping.
    - The cache is divided into "sets," and each main memory block maps to a specific set (like direct mapping).
    - Within that set, the memory block can be placed in *any* of the available cache lines (like fully associative mapping, but only within the set).
    - **Advantage:** Reduces conflict misses compared to direct mapping while being less complex and costly than fully associative mapping. Common examples include 2-way, 4-way, or 8-way set-associative caches.



$$\text{Cache Hit Rate} = \left(\frac{\text{Number of Cache Hits}}{\text{Total Number of Requests (Hits + Misses)}} \right) \times 100\%$$

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

Section D:

Pipelining

- **Concept**: An implementation technique where multiple instructions are overlapped in execution, like an assembly line. Divides instruction processing into stages.
- **Goal**: Improve instruction throughput (instructions per unit time), not latency of a single instruction.
- **MIPS 5-Stage Pipeline**:
  1. **IF**: Instruction Fetch.
  2. **ID**: Instruction Decode and Register Fetch.
  3. **EX**: Execute / Address Calculation.
  4. **MEM**: Memory Access.
  5. **WB**: Write Back.
- **Pipeline Registers**: Inserted between stages to hold data and control signals for the instruction in that stage (e.g., IF/ID, ID/EX, EX/MEM, MEM/WB registers). Control signals are passed down the pipeline with the instruction.

2. Pipeline Hazards

- Situations that prevent the next instruction in the pipeline from executing during its designated clock cycle.
- **Data Hazards**: An instruction depends on the result of a previous instruction that is still in the pipeline.
  - **Read After Write (RAW)**: Instruction tries to read a register before a previous instruction writes to it.
    - Example: add $s0, $t0, $t1 followed by sub $t2, $s0, $t3. sub needs $s0 before add writes it back in WB stage.
  - **Write After Read (WAR)**: Instruction tries to write to a register before a previous instruction reads it. (Less common in simple MIPS pipeline).
  - **Write After Write (WAW)**: Instruction tries to write to a register before a previous instruction writes to it. (Can happen with out-of-order execution).
- **Control Hazards (Branch Hazards)**: Occur when the pipeline makes a wrong decision on a branch prediction or when the branch decision is not available when needed. The instruction fetch unit doesn't know which instruction to fetch next.

3. Hazard Solutions

- **Stalling (Bubbles)**: Pause the pipeline by inserting NOP (no-operation) instructions until the hazard is resolved. Reduces performance.
- **Forwarding (Bypassing)**: Route data directly from the output of one pipeline stage (e.g., EX/MEM or MEM/WB registers) to the input of an earlier stage (e.g., EX stage ALU input) for a subsequent dependent instruction.
  - Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage.
- **Load-Use Data Hazard Stall**: Even with forwarding, if a load instruction (lw) is immediately followed by an instruction that uses the loaded value, a 1-cycle stall is often needed because the data is only available after the MEM stage.
- **Branch Hazard Solutions**:
  - **Flush/Squash**: Discard instructions fetched after a branch if the branch is taken (or mispredicted).
  - **Branch Prediction**: Guess the outcome of the branch.
    - **Static**: Always predict taken, always predict not taken, or compiler hints (e.g., predict backward branches taken for loops, forward branches not taken for ifs).
    - **Dynamic**: Hardware remembers past branch behavior (e.g., 2-bit branch predictor with states: strong not taken, weak not taken, weak taken, strong taken).
  - **Earlier Branch Decision**: Move branch decision logic to an earlier pipeline stage (e.g., ID stage instead of EX or MEM) to reduce the penalty.

4. Performance

- **CPI (Cycles Per Instruction)**: Average number of clock cycles per instruction.
- Ideal pipelined CPI = 1 (if no hazards).
- Performance comparison (example for 100 billion instructions from SPECINT2000 benchmark):
  - Single-cycle: 92.5 seconds.
  - Multi-cycle: 133.9 seconds.
  - Pipelined (with stalls for load-use and branch misprediction): 63.3 seconds.

5. Advanced Microarchitecture

- **Superscalar Processor**: Issues multiple instructions per clock cycle by having multiple execution units (e.g., multiple ALUs). Can process instructions in parallel if no dependencies.
- **Out-of-Order (OoO) Execution**: Allows instructions to execute in an order different from the program order to improve utilization of execution units, while maintaining program correctness.
    - Requires **Register Renaming**: Maps architectural registers (visible to programmer) to a larger set of physical (scratch) registers to eliminate WAR and WAW hazards.
- **Speculative Execution**: Executing code before it is known if it is even going to be needed, such as instructions after a predicted branch. If prediction is wrong, results are discarded. Handles exceptions with "poison bits" or by reordering commit.
- **Flynn's Taxonomy**:
    - SISD: Single Instruction, Single Data (Uniprocessor).
    - SIMD: Single Instruction, Multiple Data (Vector processors, GPUs).
    - MISD: Multiple Instruction, Single Data (Rare).
    - MIMD: Multiple Instruction, Multiple Data (Multiprocessors, multi-core).

6. Memory Hierarchy

- **Goal**: Provide fast access to a large amount of memory at a reasonable cost by combining different types and speeds of memory.
- **Levels** (from fastest/smallest/most expensive to slowest/largest/cheapest):
    - Registers.
    - Caches (L1, L2, L3): SRAM-based. Bridge speed gap between CPU and RAM.
    - Main Memory (RAM): DRAM-based.
    - Secondary Storage (Disk/SSD): Used for virtual memory.
- **Locality of Reference**:
    - **Temporal Locality**: Recently accessed items (data or instructions) are likely to be accessed again soon.
    - **Spatial Locality**: Items near recently accessed items are likely to be accessed soon.

7. Cache

- **Cache Hit**: Requested data is found in the cache.
- **Cache Miss**: Requested data is not in the cache; must be fetched from a lower level.
- **Hit Rate**: Fraction of memory accesses found in the cache.
- **Miss Rate**: 1−Hit Rate.
- **Average Memory Access Time (AMAT)**: AMAT=Thit+MR×MP Where Thit is hit time, MR is miss rate, MP is miss penalty. For multi-level: AMATL1=TL1+MRL1×(TL2+MRL2×TMem). Example: L1 hit=1 cycle, L1 MR=5%, L2 hit=10 cycles, L2 MR=20%, Mem time=100 cycles. AMAT=1+0.05×(10+0.2×100)=1+0.05×(10+20)=1+0.05×30=1+1.5=2.5 cycles. (Mock exam D1 uses L1 hit, L1 MR, L2 access (implies L2 hit+L1 miss penalty), L2 MR, Main mem access).
- **Cache Block (Line)**: Unit of data transfer between cache and memory (e.g., multiple words). Contains data, tag, and valid bit.
- **Cache Organization**:
    - **Direct Mapped**: Each memory block maps to exactly one cache line. (Index = (BlockAddress / BlockSize) mod NumberOfCacheLines).
    - **Fully Associative**: A memory block can be placed in any cache line. Requires searching all lines.
    - **N-way Set Associative**: Cache is divided into sets. Each memory block maps to one set, and can be placed in any of the N lines (ways) within that set. (SetIndex = (BlockAddress / BlockSize) mod NumberOfSets). A 2-way set-associative cache with 4 sets can store 2×4=8 blocks.
- **Replacement Policies** (when a cache line must be evicted):
    - LRU (Least Recently Used).
    - Pseudo-LRU.
    - Random, FIFO.
- **Write Policies**:
    - **Write-Through**: Write to both cache and main memory simultaneously.
    - **Write-Back**: Write only to cache. Write to main memory later when the block is replaced (uses a "dirty bit" D to indicate if the block was modified).
- **Miss Classification**:
    - **Compulsory Misses**: First access to a block; must be brought into cache.
    - **Capacity Misses**: Cache is too small to hold all concurrently used data.
    - **Conflict Misses**: In direct-mapped or set-associative caches, too many blocks map to the same set/line, causing useful blocks to be evicted.

8. Virtual Memory (VM)

- **Purpose**: OS memory management technique allowing secondary memory (disk) to be used as if it were part of main memory (RAM). Enables running programs larger than physical memory and provides memory protection.
- **History**: Overlays were manual program segmentation; VM automated this.
- **Paging**: Divides virtual address space into fixed-size **pages** (e.g., 4KB) and physical memory into fixed-size **page frames**. Physical memory acts as a fully associative cache for virtual memory pages.
- **Address Translation**: Virtual Address (VPN + Page Offset) -> Physical Address (PFN + Page Offset).
- **Page Table**: Stores mappings between Virtual Page Numbers (VPNs) and Physical Frame Numbers (PFNs) for each process. Usually stored in main memory. Accessing it adds an extra memory access.
- **Page Fault**: Occurs if a referenced virtual page is not in physical memory. OS handles by:
  1. Finding a free physical frame (or evicting one).
  2. Loading the required page from disk (swap space) into the frame.
  3. Updating the page table.
  4. Resuming the instruction.
- **Translation Lookaside Buffer (TLB)**: A small, fast cache (often fully associative) for recently used page table entries to speed up address translation.
- **VM Replacement Policies**: LRU, FIFO, often involve a "use bit" and "dirty bit". Writing a modified (dirty) page back to disk is called "paging" or "swapping".
- **Multilevel Page Tables**: Used to reduce the size of page tables for large address spaces (e.g., a page directory pointing to page tables).
- **Segmentation**: Divides virtual address space into variable-size, logically independent **segments** (e.g., code, data, stack). Programmers are aware of segments. Allows different protection attributes per segment (e.g., execute-only code segment). Can lead to external fragmentation. Can be combined with paging.
- **Paging vs. Segmentation**: Pages are fixed size, segments are variable size. Paging is transparent to programmer, segmentation is not. Segmentation facilitates sharing and protection based on logical units.

9. Program Loading, ABI, and ELF

- **API (Application Programming Interface)**: Source-level interface, defines how software components interact.
- **ABI (Application Binary Interface)**: Low-level, machine code interface. Defines:
  - **Calling Convention**: How arguments are passed (registers, stack), return values handled, registers saved/restored.
  - Data structure layout, system call invocation.
- **ELF (Executable and Linkable Format)**: Standard binary file format for executables, object code, shared libraries on Unix-like systems.
  - **Types**: Relocatable (.o), Executable, Shared object (.so).
  - **Structure**:
    - ELF Header: Magic number, file type, machine architecture, entry point.
    - Program Header Table (Segments): Info for OS to load program into memory (e.g., .text, .data segments with virtual addresses, permissions).
    - Section Header Table (Sections): Info for linker (e.g., .text, .data, .bss, .symtab, .strtab, .rel.text).
  - **Symbols**: nm lists symbols. Undefined symbols (like puts in an object file before linking) require linking.
  - **Relocation**: Information (.rel.text, .rel.data) used by linker to patch addresses once final memory locations are known.
- **Linking**: Combines multiple relocatable object files and resolves external references to create an executable or shared library.
- **Shared Libraries (.so)**: Code shared by multiple programs. Loaded into memory once. Requires Position Independent Code (PIC) so it can be loaded at any address. LD_LIBRARY_PATH helps locate them. ldd shows dynamic dependencies.
- **Name Mangling**: C++ compilers change function names to include argument types; extern "C" prevents this for C compatibility. c++filt demangles names.

10. Processor Examples & Microarchitecture

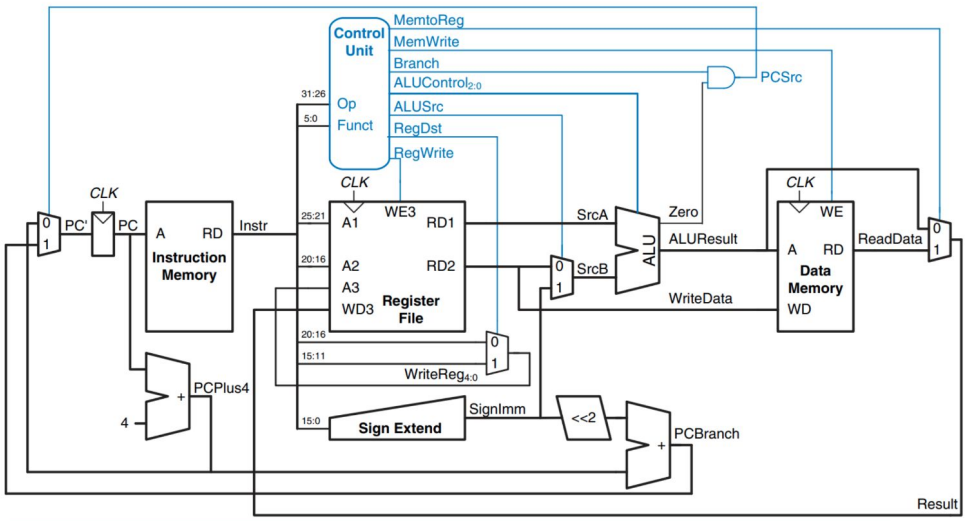
- **Microarchitecture**: The specific implementation of an Instruction Set Architecture (ISA). Includes registers, ALU, memory organization, control unit, pipelining, etc.. Different microarchitectures can implement the same ISA with different trade-offs (performance, power).
- **Intel Core i7 (Sandy Bridge example)**:
  - CISC ISA (x86) implemented on a highly pipelined RISC-like core.
  - x86 instructions are decoded into simpler micro-operations (μops).
  - Features: Deep pipelining, μop cache, out-of-order execution, register renaming (160 physical registers), speculative execution, multiple functional units (ALUs, FPUs), multi-level caches (L1 I-cache, L1 D-cache, L2, L3), store-to-load forwarding.
  - L1 D-cache is write-back.
  - ReOrder Buffer (ROB) ensures in-order retirement/commit.

- L1 is the **fastest but smallest** cache in the system—fully hardware managed, faster than L2, L3, RAM, or disk.Its latency is usually 1–2 cycles, while:
  L2 might be ~4–12 cycles
  L3 can be ~20–50 cycles
  RAM is ~100+ cycles
- L1 is often split into **two parts**:
  L1i: Instruction cache
  L1d: Data cache. This split allows parallel fetching of instructions and data.
- **ARM Cortex A9**:
  - RISC ISA (ARMv7).
  - Instructions are already like μops, simpler front-end than Core i7.
  - Features: Deep pipeline (e.g., 8-11 stages), branch prediction, split L1 I-cache and D-cache, L2 cache.
- **ATmega168 (AVR)**:
  - Simple 8-bit RISC-like processor for embedded systems.
  - Most instructions execute in 1 cycle.
  - Very short pipeline (2 stages: fetch, execute). No complex hazard detection needed due to simple pipeline and instruction set design.
  - Harvard architecture (separate address spaces and buses for data and program memory initially, though some mechanisms might allow broader addressing).

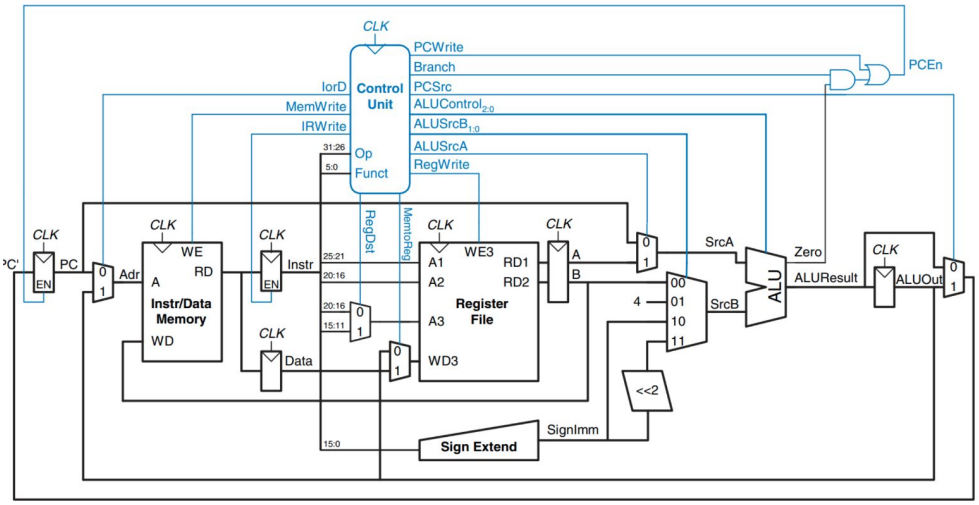
Multithreading & Multiprocessing

- **Process**: An instance of a program in execution, with its own address space.
- **Thread**: A unit of execution within a process. Threads share the process's address space but have their own stack, PC, and registers.
- **Context Switching**: OS saves state of current thread/process and restores state of another.
- **Homogeneous Multiprocessing (Symmetric Multiprocessing - SMP)**: Multiple identical processors sharing a single main memory and I/O.
- **Heterogeneous Multiprocessing (Asymmetric Multiprocessing)**: Different types of cores or specialized processors for different tasks (e.g., CPU + GPU).

Single-cycle CPU



Multi-cycle CPU



Base format for N2T

```
CHIP And {
        IN a, b;
        OUT out;

        PARTS:
```

OR

```
Not(in=a, out=a1);
Not(in=b, out=b1);
Nand(a=a1, b=b1, out=out);
```

AND

```
Nand(a=a, b=b, out=out1);
Not(in=out1, out=out);
```

NOT

```
Nand(a=in , b=in , out=out);
```

Mux

```
Not(in=sel, out=Nsel);
And(a=a, b=Nsel, out=o1);
And(a=b, b=sel, out=o2);
Or(a=o1, b=o2, out=out);
```

DMux

```
Not(in=sel, out=Nsel);
And(a=Nsel, b=in, out=a);
And(a=sel, b=in, out=b);
```

Half Adder

```
XOr(a=a, b=b, out=sum);
And(a=a, b=b, out=carry);
```

INC16

```
Add16(a=in, b[0]=true,
b[1..15]=false, out=out);
```

Full Adder

```
Xor(a= c, b = out1, out =
sum);
Xor(a= a, b= b, out= out1);
And(a= c, b= out1, out=
out3);
And(a= a, b= b, out=out2);

Or(a= out3, b= out2, out=
carry);;
```

XOR

```
Not(in=a, out=a1);
Not(in=b, out=b1);
And(a=a, b=b1, out=o1);
And(a=a1, b=b , out=o2);

Or(a=o1, b=o2, out=out);
```

Add16

```
FullAdder(a=a[0], b=b[0],
c=false, sum=out[0],
carry=c0);
…
FullAdder(a=a[15], b=b[15],
c=c14, sum=out[15],
carry=c15);
```

CPU

```
Not(in=instruction[15], out=isAInstruction);
Not(in=isAInstruction, out=isCInstruction);
Mux16(a=instruction, b=aluOutput, sel=isCInstruction, out=aRegisterInput);
And(a=instruction[5], b=isCInstruction, out=storeALUResultInARegister);
Or(a=storeALUResultInARegister, b=isAInstruction, out=aRegisterLoad);
ARegister(in=aRegisterInput, load=aRegisterLoad, out=aRegisterOutput,
out[0..14]=addressM);
Mux16(a=aRegisterOutput, b=inM, sel=instruction[12], out=aluYInput);
And(a=instruction[4], b=isCInstruction, out=dRegisterLoad);
DRegister(in=aluOutput, load=dRegisterLoad, out=dRegisterOutput);
ALU(x=dRegisterOutput, y=aluYInput,
      zx=instruction[11], nx=instruction[10],
      zy=instruction[9],  ny=instruction[8],
      f=instruction[7],   no=instruction[6],
      out=aluOutput, out=outM, zr=aluZero, ng=aluNegative);
And(a=instruction[3], b=isCInstruction, out=writeM);
Not(in=aluZero, out=isNonZero);
Not(in=aluNegative, out=isNonNegative);
And(a=isNonZero, b=isNonNegative, out=isPositive);
And(a=instruction[2], b=aluNegative, out=jumpIfNegative);
And(a=instruction[1], b=aluZero, out=jumpIfZero);
And(a=instruction[0], b=isPositive, out=jumpIfPositive);
Or(a=jumpIfNegative, b=jumpIfZero, out=jumpIfNegOrZero);
Or(a=jumpIfNegOrZero, b=jumpIfPositive, out=shouldJump);
And(a=shouldJump, b=isCInstruction, out=loadPC);
PC(in=aRegisterOutput, load=loadPC, inc=true, reset=reset, out[0..14]=pc);
```

Bit

```
Mux(a=oldOut, b=in, sel=load, out=o1);
DFF(in=o1, out=oldOut, out=out);
```

PC

```
Inc16(in=inMain, out=incMain);
Mux16(a=inMain, b=incMain, sel=inc, out=o1);
Mux16(a=o1, b=in, sel=load, out=o2);
Mux16(a=o2, b=false, sel=reset, out=o3);
Register(in=o3, load=true, out=inMain, out=out);
```

Register

```
Bit(in=in[0], load=load, out=out[0]);
…
Bit(in=in[15], load=load, out=out[15]);
```

Memory

```
DMux(in=load, sel=address[14], a=rLoad, b=sLoad);
RAM16K(in=in, address=address[0..13], load=rLoad, out=ramOut);
Screen(in=in, address=address[0..12], load=sLoad, out=screenOut);
Keyboard(out=keyboardOut);
Mux4Way16(a=ramOut, b=ramOut, c=screenOut, d=keyboardOut, sel=address[13..14],
out=out);
```

RAM8

```
DMux8Way(in= load, sel= address, a= in1, b= in2, c= in3, d= in4, e= in5, f= in6, g= in7, h= in8);
Register(in= in, load= in1, out= o1);
Register(in= in, load= in2, out= o2);
Register(in= in, load= in3, out= o3);
Register(in= in, load= in4, out= o4);
Register(in= in, load= in5, out= o5);
Register(in= in, load= in6, out= o6);
Register(in= in, load= in7, out= o7);
Register(in= in, load= in8, out= o8);
Mux8Way16(a= o1, b= o2, c= o3, d= o4, e= o5, f= o6, g= o7, h= o8, sel= address, out= out);
```

ALU

```
    Mux16(a= true, b= false, sel= zx, out= zerox);
    And16(a= zerox, b= x, out= zxdone);
    Mux16(a= false, b= true, sel= nx, out= negatex);
    Xor16(a= zxdone, b= negatex, out= nxdone);

    Mux16(a= true, b= false, sel= zy, out= zeroy);
    And16(a= zeroy, b= y, out= zydone);
    Mux16(a= false, b= true, sel= ny, out= negatey);
    Xor16(a= zydone, b= negatey, out= nydone);

    Add16(a = nxdone, b = nydone, out = added);
    And16(a= nxdone, b= nydone, out= anded);
    Mux16(a= anded, b= added, sel= f, out= functionresult);
    Mux16(a= false, b= true, sel= no, out= negate);
    Xor16(a= functionresult, b= negate, out= out);
    Xor16(a= functionresult, b= negate, out[15]= ng);
    Xor16(a= functionresult, b= negate, out[0..7]= tempout1, out[8..15] =
tempout2);
    Or(a= tempout1, b= tempout2, out= i1);
      Or8Way(in=tempout1, out=lowerBitsNonZero);
      Or8Way(in=tempout2, out=upperBitsNonZero);
      Or(a=lowerBitsNonZero, b=upperBitsNonZero, out=isNonZero);
      Not(in=isNonZero, out=zr);
```