Unit 6- Microprocessor Design

INTRODUCTION

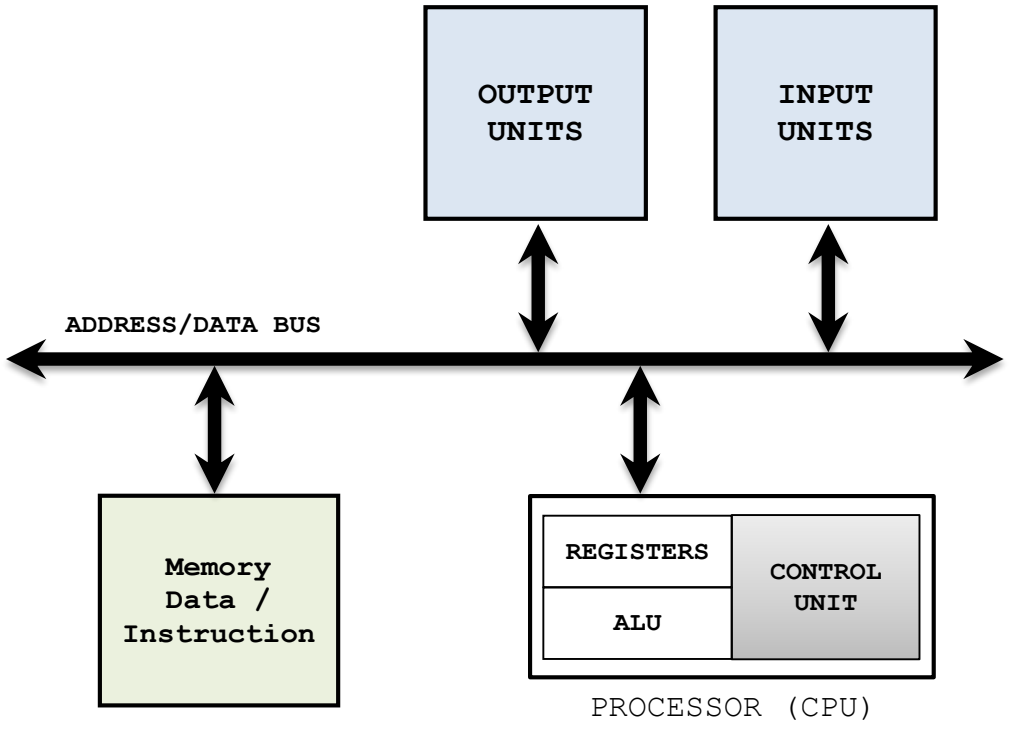
- Abstraction Layers in Computer Systems Design: Transistor Circuits → Logic Gates → **Register Transfers** → Microarchitecture → Instruction Set Architecture → Operating Systems → Programming Languages → Algorithm
- Typical devices used to implement digital systems (they can be implemented with a hardware-description language):
  - ✓ ASICs, FPGAs: For dedicated hardware implementation. It requires highly specialized design.
  - ✓ General-Purpose Microprocessors, Microcontrollers (e.g. embedded μC). It requires software development.
  - ✓ Specialized uPs: PDSPs (programmable digital signal processor). It requires specialized software development.
- ASICs or uPs? Performance vs. flexibility. ASIC design requires high development cost, not reprogrammable.
- FPGAs: Intermediate option between ASICs and uP. Not commonly used for processor implementation. Operating frequencies can be relatively low compared to uP, but can achieve higher performance for specific tasks. They are reconfigurable.
- PSoCs (Programmable System-on-Chip). They integrate reconfigurable logic (like an FPGA), a hard-wired microprocessor, and peripherals. With proper software/hardware co-design, high performance solutions can be attained.

COMPUTER HARDWARE ORGANIZATION

- **General-purpose Digital Computer**: Usually called 'Computer'. It is a digital system that can follow a stored sequence of instructions, called a program, that operates on data.
  - ✓ The user can specify and modify the program and/or the data according to their specific needs.
  - ✓ As a result of this flexibility, general-purpose digital computers can perform a variety of information-processing tasks, ranging over a very wide spectrum of applications.
  - ✓ The digital computer is thus a highly general and very flexible digital system.

- **Computer Specification**: It is the description of its appearance to a programmer at the lowest level: the *Instruction Set Architecture (ISA)*. From the ISA, a high-level description of the hardware to implement the computer (i.e., the *computer architecture*) is formulated.

- **Computer**: Processor + I/O + Memory
  - ✓ Memory: It stores programs as well as input, output, and intermediate data.
  - ✓ Central Processing Unit (CPU): It sequentially executes the instructions in memory (the program) by performing arithmetic and other data-processing operations.
  - ✓ I/O Units: A digital computer can accommodate many different input and output devices, e.g.: DVD drives, USB flash drives, printers, LCDs, keyboards.



CENTRAL PROCESSING UNIT (CPU)

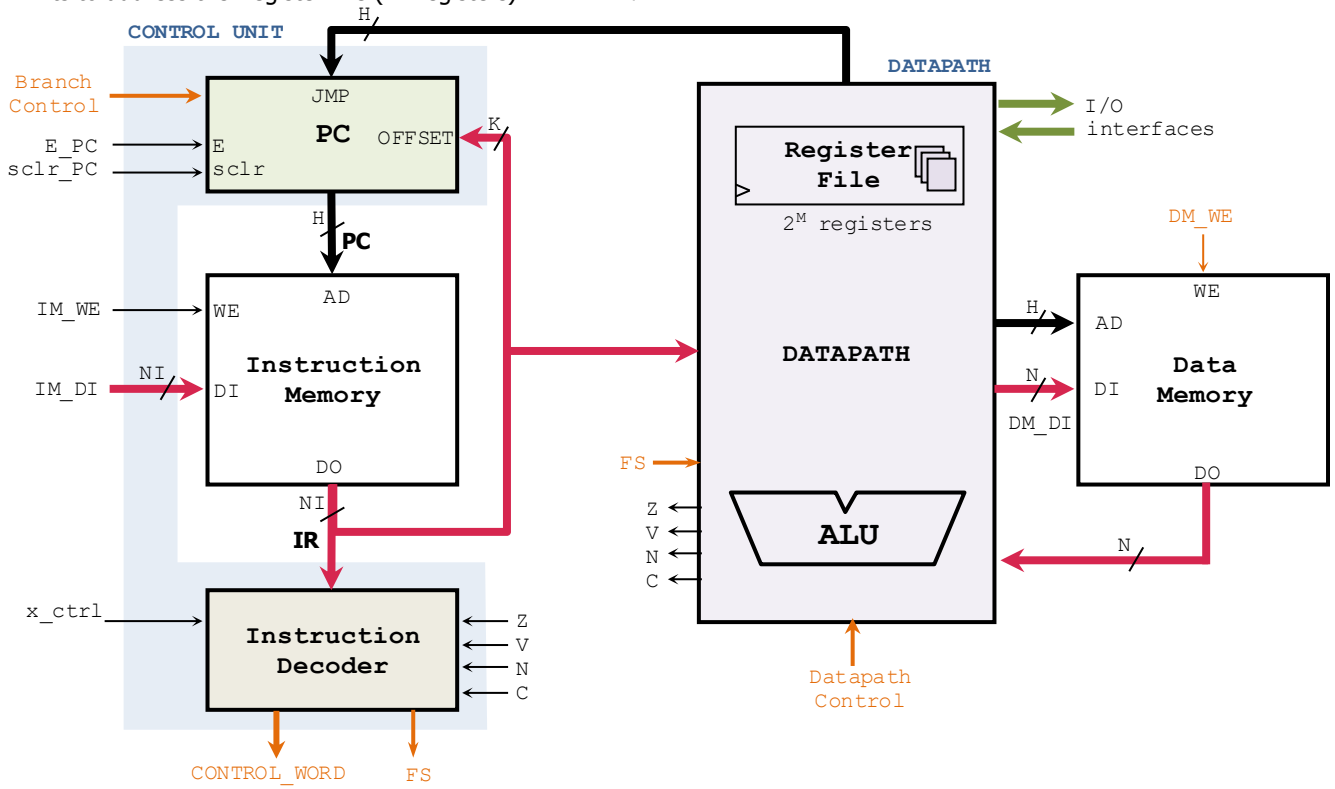
- Also called Processor. It consists of a Datapath and Control Unit.
  - ✓ **Datapath**:
    - **Register File** (set of Registers): They hold data and memory address values during the execution of an instruction.
    - **Arithmetic Logic Unit (ALU)**: Shared operation unit that performs arithmetic (e.g., addition, subtraction, division) and bit-wise logic (e.g., AND, OR, operations).
  - ✓ **Control Unit:** It controls operations performed on the Datapath and other components (e.g. memory). It interprets the instructions and executes them. Instructions are read from memory. To execute a particular instruction, this unit asserts specific signals at certain times to control the registers, ALU, memories and ancillary logic. A Control Unit usually includes:
    - Program Counter (PC): During program execution, it provides the address of the instruction being executed. It can increase the address as well as change the sequence of operations using decisions based on status information.
    - Instruction Decoder (ID): It reads the instructions and generates control signals to the datapath and other components. It is usually implemented as a combinational circuit (single-cycle computers) or as a large Finite State Machine (FSM) with ancillary logic (multi-cycle computers).
- Complex CPU: Multiple control units and datapaths.

Harvard vs. Von Neumann

Harvard:	▫ Instruction memory and Data memory ▫ Operands usually placed in registers in the CPU: register-to-register architecture
Von Neumann:	▫ One memory for both instruction and data ▫ Operands placed in an accumulator register or in the instruction memory: register-memory architecture

GENERIC CPU MODEL

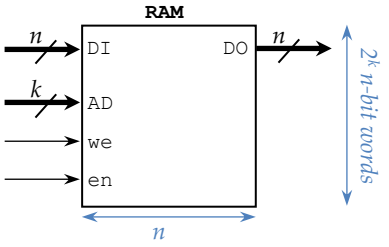
- The figure depicts a generic model for a CPU with typical components. The Control Unit includes the Program Counter (PC) and the Instruction Decoder (ID). The Datapath includes a Register File and an ALU. Instruction and Data Memories are usually included. A specific CPU might not have all the components or connections, or it might include more components.
- Program Counter (PC): It has a branch control mechanism to increment the PC, assign an arbitrary value (jump/ branch), or to apply an address offset. The jump/branch address and offset are latched from the instruction itself or from the datapath. In the figure, the instruction register (IR) goes to the offset address, while the Datapath generates the jump address. But it can be the other way around, or the PC might not include an offset or jump address.
- Instruction Decoder (ID): It generates control bits (orange-colored signals) for the Datapath, PC, and Data Memory.
- Instruction Memory (IM): It generates the instructions to be executed. The output is called the Instruction Register (IR).
- This CPU requires an extra circuitry that: i) enables the execution of PC (E_PC, sclr_PC), ii) controls Instruction Memory (IM) loading, and iii) enables the Instruction Decoder.
- Model Parameters: H: Memory address word size. K: Address Offset Size. N: Data word size. NI: Instruction Word Size. M: Bits to address the Register File (2^M registers). $K \leq H \leq N$.



MEMORY OVERVIEW

RANDOM ACCESS MEMORY (RAM)

- Access to words from a desired location take the same time regardless of the location, hence the name RAM.
- Number of words: 2^k n-bit words ($2^k \times n$ memory). Depth = $m = 2^k$, width = n.
- I/O description:
  - ✓ `DI`: n-bit input data.
  - ✓ `DO`: n-bit output data.
  - ✓ `AD`: k-bit address. The addresses range from 0 to $2^k - 1$.
  - ✓ `we`: write enable. Also called `R/W` (sometimes this signal is active-low: $R/\overline{W}$)
  - ✓ `en`: enable. Other terms are `CS` (chip select) or `E`.



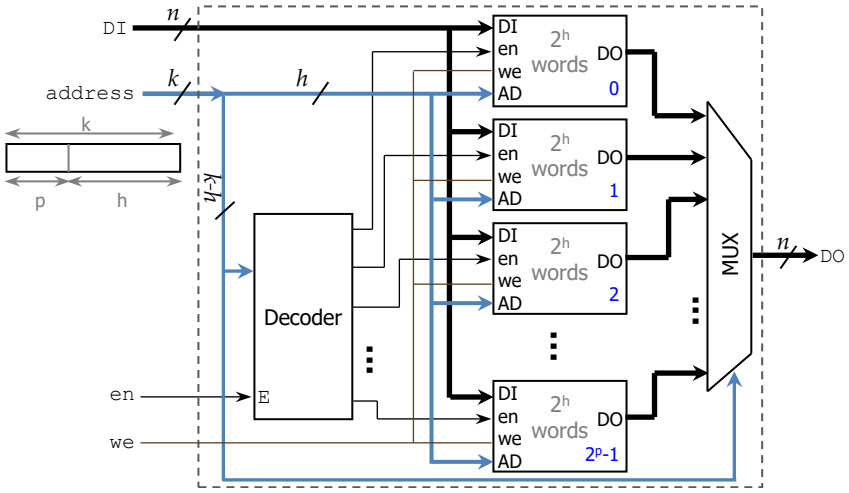
- Operation:

en	we	Action
0	X	no read or write
1	0	read word from address pointed by AD
1	1	write on address pointed by AD

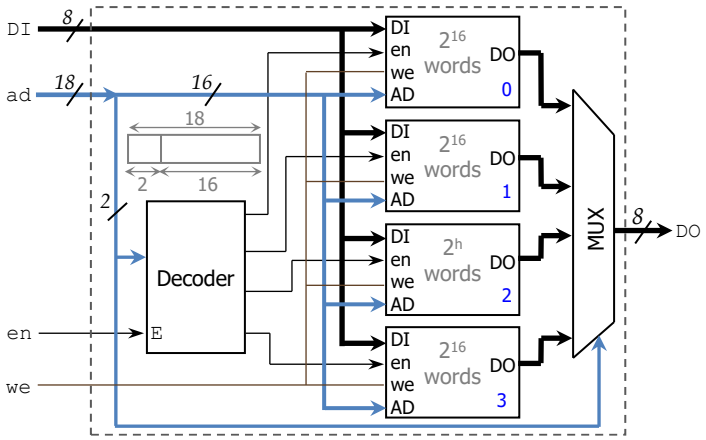
  - ✓ In simple memory designs, we often tie `en=1`, so that we always read or write. Here, `we` is often renamed as `wr_rd`.

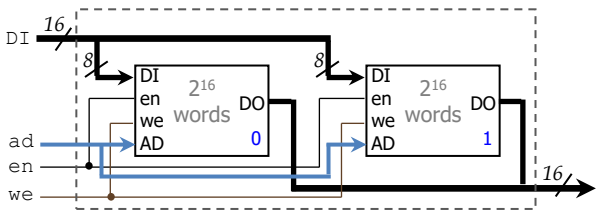
MEMORY DECODING

- We can group memory blocks to build a larger memory. For a memory with 2^k n-bit words, if an individual memory block has 2^h words, we then need $2^{k-h} = 2^p$ memory blocks.
- Conceptual implementation:
  - ✓ Data write: a decoder can be used to select the proper block to write on.
  - ✓ Data read: a multiplexor (or 3-state buffer) can be used to select which memory block output to read from.
- The k-bit address of the memory can be divided into $p + h$ bits.
  - ✓ p: bits required to select the individual memory blocks (to write or read).
  - ✓ h: bits required to select the words from an individual memory block.



- **Example**: 256K×8 memory out of 4 64×8 memory blocks. The 18-bit address of the 256K×8 memory is divided into 2 bits (to select the memory block to read/write data) and 16 bits (to select an individual address from a memory block).
  - ✓ Note that for every bit we increase in the address, the number of words doubles.
  - ✓ We can also concatenate the data inputs (and data outputs) of several memory blocks to create a memory with an increased word size.
    - Example: 256Kx16 memory (`address`, `we`, and `en` inputs are shared amont all the memory bocks).

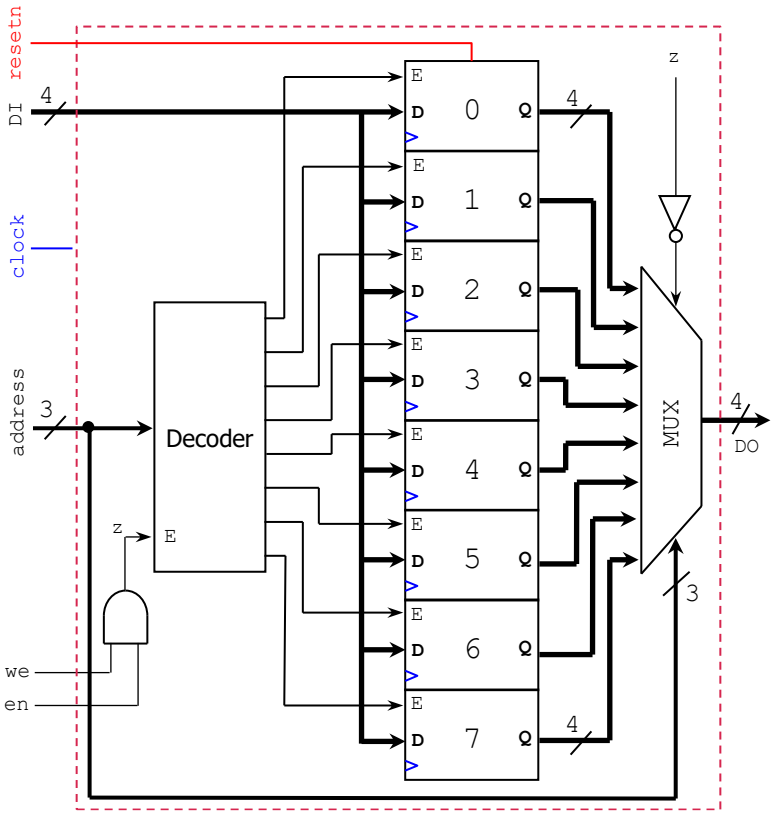




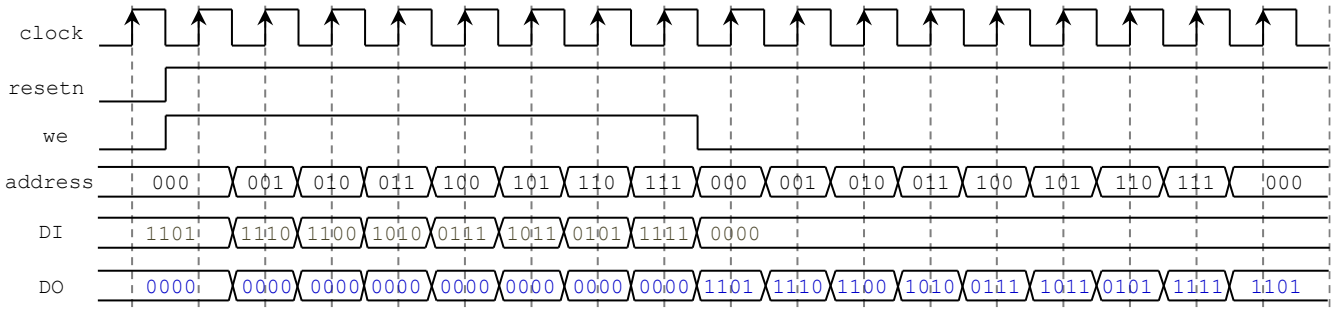
RAM IMPLEMENTATION

Register-based:

- These designs can be described at RTL (on FPGAs or ASICs).
- Write operation: A given input word is stored on the desired address (register) right after the clock edge.
  - ✓ There are different approaches for the behavior of the output data when writing. Simplest one: set all outputs to 0's.
- Read operation: Data is available as soon as the address of the desired word is fed.



- The figure shows a memory with n=4, k=3. A timing diagram is shown, where we tie `en=1`.
  - ✓ Sometimes an output register is added (this delays output for 1 clock cycle).
  - ✓ In many applications, it is common to set `en=1`. Here, `we` controls reading/writing.
- VHDL parametric code: RAM_emul.vhd. (it has `wr_rd = we`; `en`: not used)
- This straightforward implementation is not used in real-world applications (unless the memory requirements are very small): flip flops are very expensive resources, also for every extra address bit the decoder and mux sizes grow exponentially.

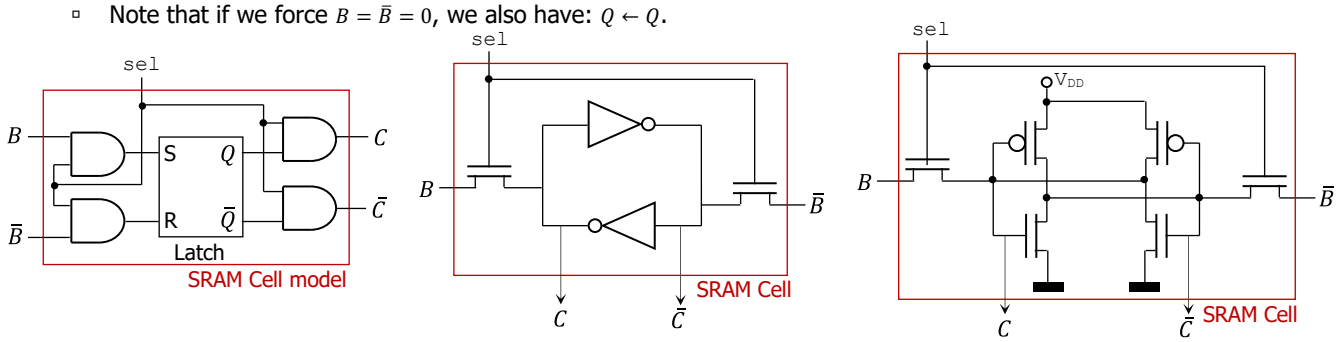


Static RAM (SRAM)

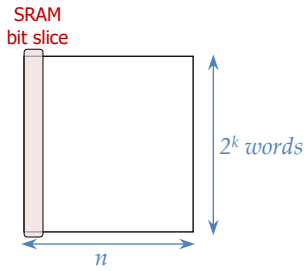
- Data is stored in latches. Data valid as long as power is applied. Compared to other memory technology, SRAMs are easier to use, they feature shorter read/write cycles, and require no refresh.
- **One-bit implementation**: the figure shows a SRAM cell logic model, a SRAM cell implementation with pass transistors and NOT gates, and a SRAM cell implementation where the NOT gates are implemented with CMOS technology.
  - ✓ FPGAs use SRAM-based technology to store the bitstream. BlockRAMs are also based on SRAM technology.
  - ✓ RAM Cell logic model functionality:

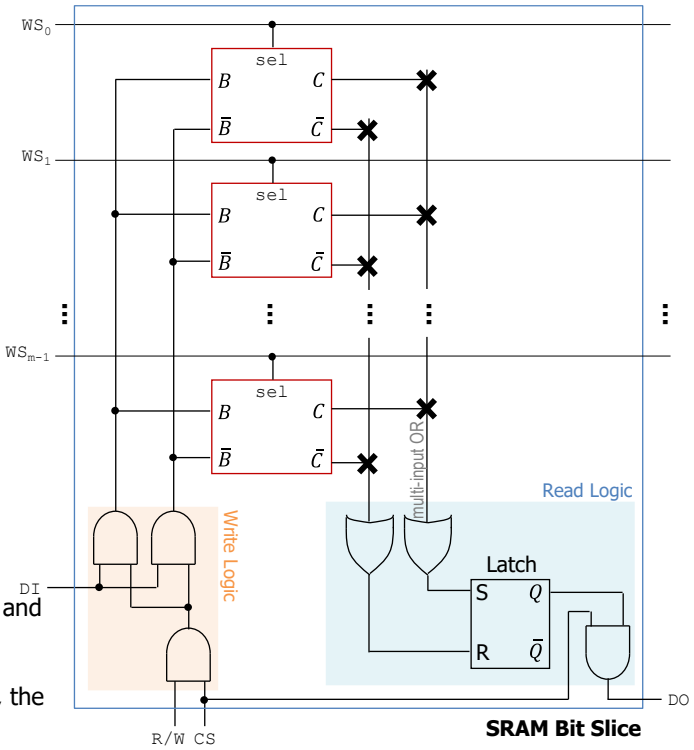
sel	Latch action	C	$\bar{C}$
0	$Q \leftarrow Q$	0	0
1	$Q \leftarrow B$	Q	$\bar{Q}$

    - Note that if we force $B = \bar{B} = 0$, we also have: $Q \leftarrow Q$.



- **SRAM implementation:** We first implement a 'bit slice', i.e., a column of the SRAM.
  - ✓ RAM bit slice: $m = 2^k$



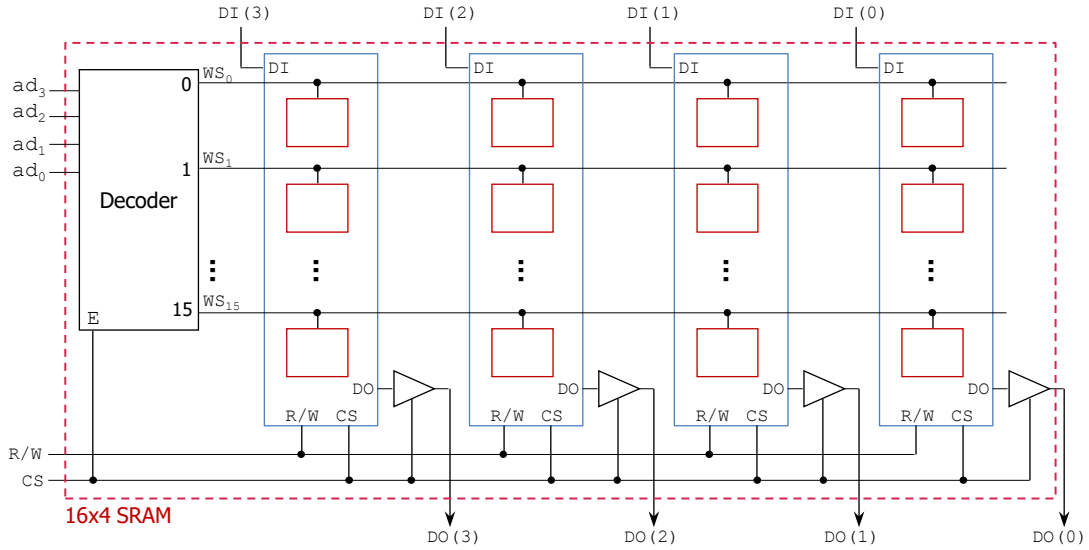


  - Write logic:

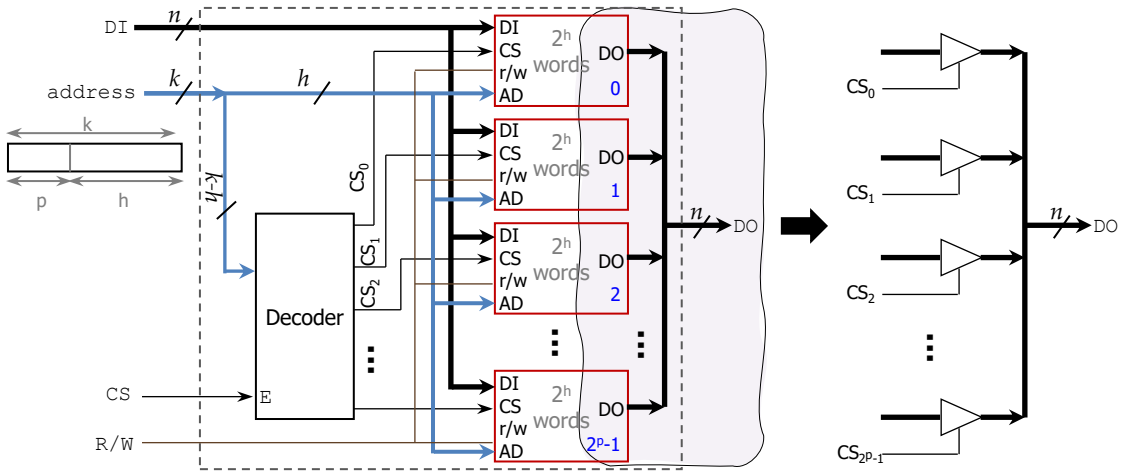
CS	R/W	B	$\bar{B}$	Comments
0	X	0	0	DO=0 (invalid)
1	0	0	0	Data kept on SRAM cells
1	1	DI	$\overline{DI}$	Can write on an SRAM cell

  - Read logic:
    - If `sel=0` for all cells, then $C = \bar{C} = 0$ for all cells and the SR latch keeps its value.
    - If `CS=0`, then `DO=0` (invalid output).
    - If `CS=1`, only one cell should have `sel=1`. Here, the SR Latch stores the cell value (C) and `DO=`C.

  - ✓ $2^k \times n$ words SRAM: It can be implemented with a group of SRAM bit slices. For example: 16x4 SRAM (n=4, k=4).

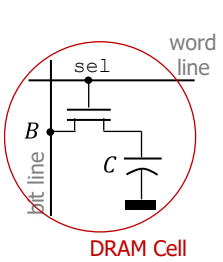


  - 3-state output buffers: They are included to optimize implementation of larger SRAMs. If we want to build a larger SRAM out of smaller SRAMs (see memory decoding), we then do not need to use a large multiplexor:

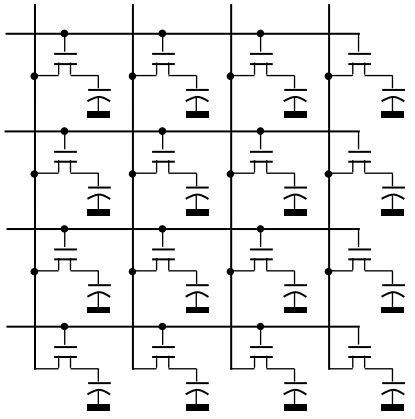


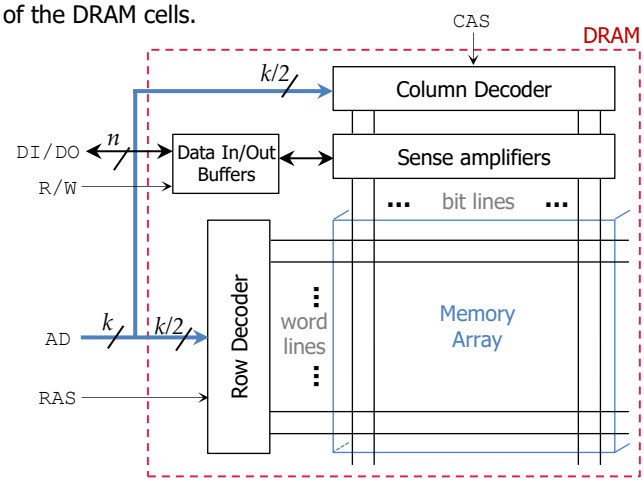
Dynamic RAM (DRAM)

- Data is stored as electric charges on capacitors. The storage of information is only temporary: data tends to discharge over time, and it must be periodically recharged by refreshing the DRAM cells (this is a refresh cycle).
- DRAMs provide high storage capacity at low cost, but they are slower than SRAMs. Also, the design is quite more challenging.
- The figure depicts a DRAM cell, a 16x1 DRAM cell array arranged as a 2D array (4x4), and a block diagram of a DRAM.
  - ✓ DRAM cell: If `sel=0`, the transistor is open, and the capacitor charge remains roughly fixed. If `sel=1`, the transistor is closed, and charge can flow into and out of the capacitor from the bit line.
    - Writing (from bitline to capacitor): the value of the bit line (B) is stored on the capacitor (by charging or discharging).
    - Reading (from capacitor to bitline): the electric charge in the capacitor is applied onto the bit line.
      - The charge in the capacitors are low-power signals. Here, we need sense amplifiers, whose role is to sense these signals and amplify them so that data can be interpreted properly by logic outside the memory.
      - A read operation depletes the charge in a cell, destroying the data. To restore it, a sense amplifier must immediately write it back in the cell by applying a voltage to it, recharging the capacitor (this is called refresh).
  - ✓ 16x1 DRAM cell array: Instead of building a 16x1 DRAM Bit Slice, a more efficient approach is to arrange the 16 cells into a 4x4 array. This separates the address into a row address and a column address (requires row/column decoders).
    - This row/column address separation technique can be applied to a $2^k \times 1$ memory, dividing into a $2^{k/2} \times 2^{k/2}$ array.
    - We can arrange a group of these 4x4 arrays (called banks) in parallel in order to build a 16×n DRAM cell array.
  - ✓ DRAM block diagram: It depicts typical components found in a DRAM: row and column decoders, sense amplifiers, and memory array (collection of n 2D arrays of size $2^{k/2} \times 2^{k/2}$, this implements a $2^k \times n$ DRAM).
    - Even with no reading, data in the capacitor discharges over time. So, we need to periodically (e.g. every 64 ms) execute a refresh cycle (refresh all DRAM cells). During a refresh cycle, no reads or writes can occur. This needs additional logic to generate the addresses of all the rows of the DRAM cells.



DRAM Cell


16x1 DRAM cell array as a 2D array (4x4)

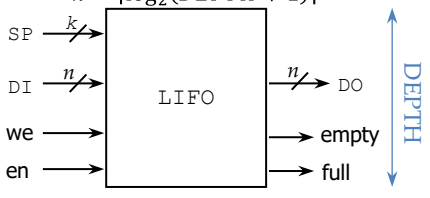


- Synchronous DRAM (SDRAM): The external interface is clocked (DRAMs have asynchronous interfaces).
- Double Data Rate SDRAM (DDR SDRAM): Like SDRAM, but the data output is provided on both the positive and negative clock edges. Voltage: 2.5 v, frequency: 133 MHz. * DDR2: 1.8v, freq: 266, 333, 400 MHz. * DDR3: 1.5v, freq: 800 MHz.

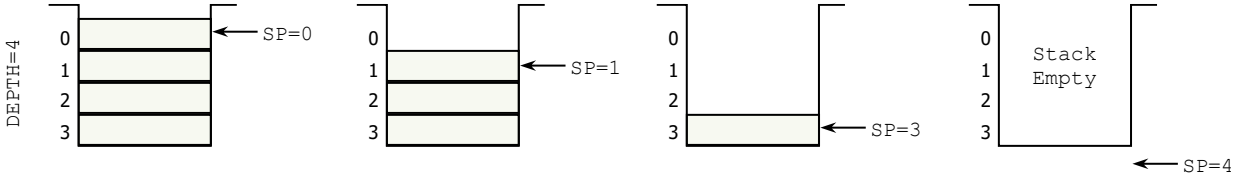
LIFO (LAST-IN, FIRST-OUT)

- The structure resembles that of a Stack. SP: Pointer to the Top of the Stack (TSP).

$k = \lceil \log_2(DEPTH + 1) \rceil$

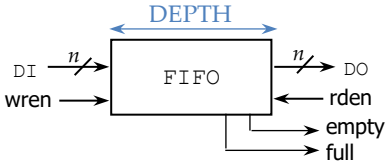


Operation				
* At power-up: SP ← DEPTH (Stack is empty)				
If en = '1':	▫ If we = '1':	SP ← SP-1	ST[SP] ← DI	
	▫ If we = '0':	DO ← ST[SP]	SP ← SP+1	
else:	DO ← ST[SP]			
Flags:	empty = 1 if SP=DEPTH, else 0		full = 1 if SP=0, else 0	



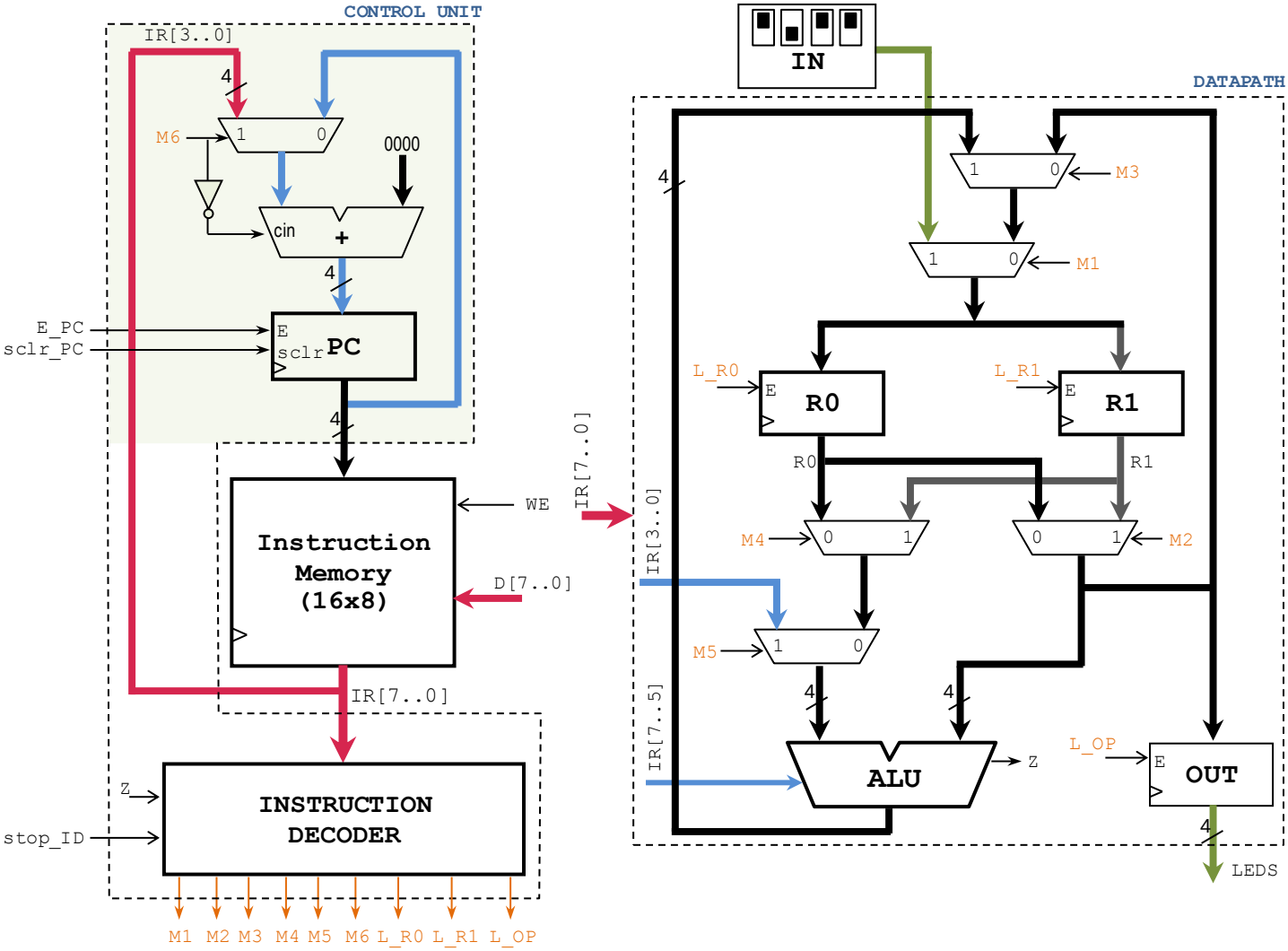
FIFO (FIRST-IN, FIRST-OUT)

- Structure that is useful to implement queues and to buffer large amounts of data.
- Synchronous FIFO: data written/read at the same clock rate.
- Asynchronous FIFO: write and read clocks can be different. This is an ASIC design (not an RTL) that is useful to pass data between different clock domains.



SINGLE-CYCLE HARDWIRED CONTROL – VBC (VERY BASIC COMPUTER)

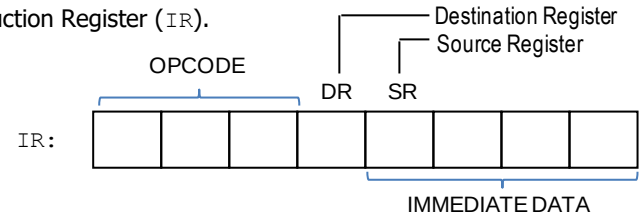
- This is a simple microprocessor where instructions are processed in one clock cycle.
- Only one instruction memory. No data memory. Data can be loaded onto the ALU on one clock cycle.
- Instruction Memory: Implemented as an array of registers. When reading, output appears as soon as address is ready.
- Instruction Decoder: the 'stop_ID' external signal makes sure that the ID outputs are '0' so that nothing gets updated.
- Note how this detailed figure fits into the Generic CPU model.



- **Register File:** `R0` (register 0, 4 bits), `R1` (register 1, 4 bits).
- **ALU:** 4-bit operations. Thus, we have up to 8 different operations.
- **Program Counter (PC)**: To execute the instructions sequentially (one after the other), we must provide the memory address of the instruction to be executed. In a computer, this address comes from a register called PC.
- **Instruction Decoder**: Converts instructions into control bits. This is a combinational circuit.
- **Instruction Memory**: Stores up to 16 8-bit instructions
- **Other Registers:** `OUT` (output register, 4 bits), `PC` (program counter, 4 bits), `IR` (instruction register, 8 bits).

INSTRUCTION SET

- Instructions are specified by the Instruction Register (`IR`).



`DR=0` ⇒ `R0` is the Destination register, `DR=1` ⇒ `R1` is the Destination register.
`SR=0` ⇒ `R0` is the Source register, `SR=1` ⇒ `R1` is the Source register.

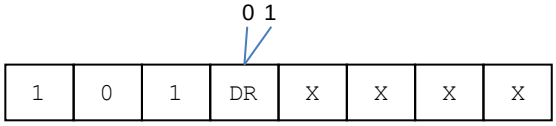
OPCODE (IR[7..5])	Instruction	Operation Description
000	MOV DR, SR	DR ← SR
001	LOADI DR, DATA	DR ← DATA, DATA = IR[3..0]
010	ADD DR, SR	DR ← DR + SR
011	ADDI DR, DATA	DR ← DR + DATA, DATA = IR[3..0]
100	SR0 DR, SR	DR ← 0&SR[3..1]
101	IN DR	DR ← IN
110	OUT DR	OUT ← DR
111	JNZ DR, ADDRESS	PC ← PC + 1 if DR=0 PC ← IR[3..0] if DR≠0 * ADDRESS = IR[3..0]

- opcode: IR[7..5]: This is the *operation code* of an instruction. This group of bits specifies an operation (such as add, subtract, shift, complement in the ALU). If it has m bits, there can be up to 2^m distinct instructions.
- Immediate Data: IR[3..0]. This is called an immediate operand since it is immediately available in the instruction.

INSTRUCTION DECODER

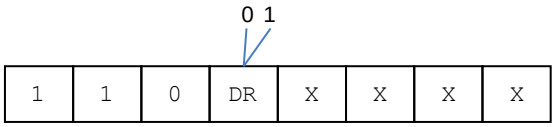
- This component is in charge of issuing control signals for the proper execution of instructions. The inputs to this circuit are the Instruction Register (IR) and the Z flag. The outputs are all the control signals: M1, M2, M3, M4, M5, M6, L_R0, L_R1, L_OP. Note that the Function Select (FS) output to the ALU is directly generated by IR[7..5].
- Also, if stop_ID=1, the following signals must be set to '0': register enables in the Datapath (L_OP, L_R0, L_R1), and the PC control signal M6. This is useful to pause execution of a program (PC and Datapath are not updated).
- This is a combinational circuit. The I/O relationship depends on how each instruction is defined.

✓ IN DR: DR grabs the contents from the input



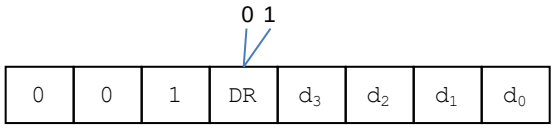
IN R0: 1010XXXX ⇒ M1 ← 1, L_R0 ← 1, M6 ← 0
IN R1: 1011XXXX ⇒ M1 ← 1, L_R1 ← 1, M6 ← 0

✓ OUT DR: Places the contents of DR on the output register



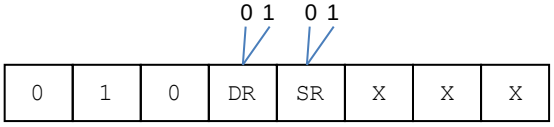
OUT R0: 1100XXXX ⇒ M2 ← 0, L_OP ← 1, M6 ← 0
OUT R1: 1101XXXX ⇒ M2 ← 1, L_OP ← 1, M6 ← 0

✓ LOADI DR, DATA: Copies immediate DATA onto DR



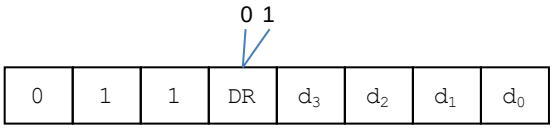
LOADI R0, DATA: $0010d_3d_2d_1d_0$ ⇒ M5 ← 1, M3 ← 1, M1 ← 0, L_R0 ← 1, M6 ← 0
LOADI R1, DATA: $0011d_3d_2d_1d_0$ ⇒ M5 ← 1, M3 ← 1, M1 ← 0, L_R1 ← 1, M6 ← 0

✓ ADD DR, SR: Adds SR and DR, and copies the result onto DR



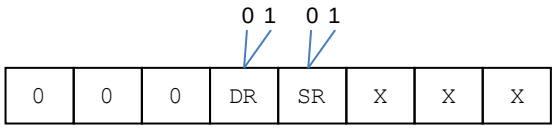
ADD R0,R0: 01000XXX ⇒ M4←0, M5←0, M2←0, M3←1, M1←0, L_R0←1, M6←0
ADD R0,R1: 01001XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R0←1, M6←0
ADD R1,R0: 01010XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0
ADD R1,R1: 01011XXX ⇒ M4←1, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0

✓ `ADDI DR, DATA:` Adds immediate `DATA` and `DR`, and copies the result onto `DR`



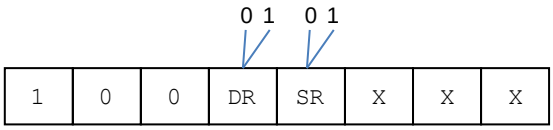
`ADDI R0, DATA:` $0110d_3d_2d_1d_0 \Rightarrow$ `M2 ← 0, M5 ← 1, M3 ← 1, M1 ← 0, L_R0 ← 1, M6 ← 0`
`ADDI R1, DATA:` $0111d_3d_2d_1d_0 \Rightarrow$ `M2 ← 1, M5 ← 1, M3 ← 1, M1 ← 0, L_R1 ← 1, M6 ← 0`

✓ `MOV DR, SR:` Copies the contents of `SR` onto `DR`



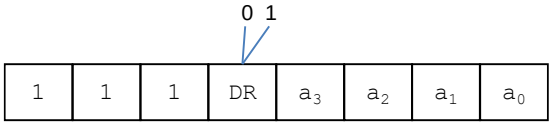

```
MOV R0, R0: 00000XXX ⇒ M2 ← 0, M3 ← 0, M1 ← 0, L_R0 ← 1, M6 ← 0
MOV R1, R1: 00011XXX ⇒ M2 ← 1, M3 ← 0, M1 ← 0, L_R1 ← 1, M6 ← 0
MOV R0, R1: 00001XXX ⇒ M2 ← 1, M3 ← 0, M1 ← 0, L_R0 ← 1, M6 ← 0
MOV R1, R0: 00010XXX ⇒ M2 ← 0, M3 ← 0, M1 ← 0, L_R1 ← 1, M6 ← 0
"MOV R0,R0", "MOV R1,R1"(can be used as NOP instruction)
```

✓ `SR0 DR, SR:` Shifts (to the right) the contents of `SR` and places the result onto `DR`




```
SR0 R0,R0: 10000XXX ⇒ M4←0, M5←0, M2←0, M3←1, M1←0, L_R0←1, M6←0
SR0 R0,R1: 10001XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R0←1, M6←0
SR0 R1,R0: 10010XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0
SR0 R1,R1: 10011XXX ⇒ M4←1, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0
```

✓ `JNZ DR, ADDRESS`: Jumps to a certain instruction if the DR contents ≠ 0. This is how computers implement loops.



`JNZ R0, ADDRESS:` $1110a_3a_2a_1a_0 \Rightarrow$ `M2 ← 0, M6 ← 0 if z = 1, M6 ← 1 if z = 0`
`JNZ R1, ADDRESS:` $1111a_3a_2a_1a_0 \Rightarrow$ `M2 ← 1, M6 ← 0 if z = 1, M6 ← 1 if z = 0`
`* M6 ← 0 ≡ PC ← PC + 1; M6 ← 1 ≡ PC ← IR[3..0]`

ARITHMETIC LOGIC UNIT

- With the 3-bit input selector FS, the operations performed here are very simple. For 4-bit inputs A and B as well as 4-bit output F, we have that: F=A when FS=000,001; F=A+B when FS=010,011; F=sr(A) when FS=100; and F=B when FS=111. The output `Z=1` if the result of F is all 0's, except is FS=101, 110 (since these are the IN, OUT instructions).

Example:

- Write an assembly program for a counter from 1 to 5: 1, 2, 3, 4, 5, 1, 2, 3, …. The count must be shown on the output register (OUT).

```
start: loadi R0,1      R0 ← 1
       out R0          OUT = 1
       addi R0,1       R0 ← R0 + 1 = 2
       out R0          OUT = 2
       addi R0,1       R0 ← R0 + 1 = 3
       out R0          OUT = 3
       addi R0,1       R0 ← R0 + 1 = 4
       out R0          OUT = 4
       addi R0,1       R0 ← R0 + 1 = 5
       out R0          OUT = 5
       jnz R0, start
```

Example:

- Write an assembly program for a counter from 2 to 13: 2,3,…, 13,2,3,… The count must be shown on the output register (OUT). Use labels to specify any address where your program jumps. Note that you can have only up to 16 instructions.
- Provide the contents of the Instruction Memory.

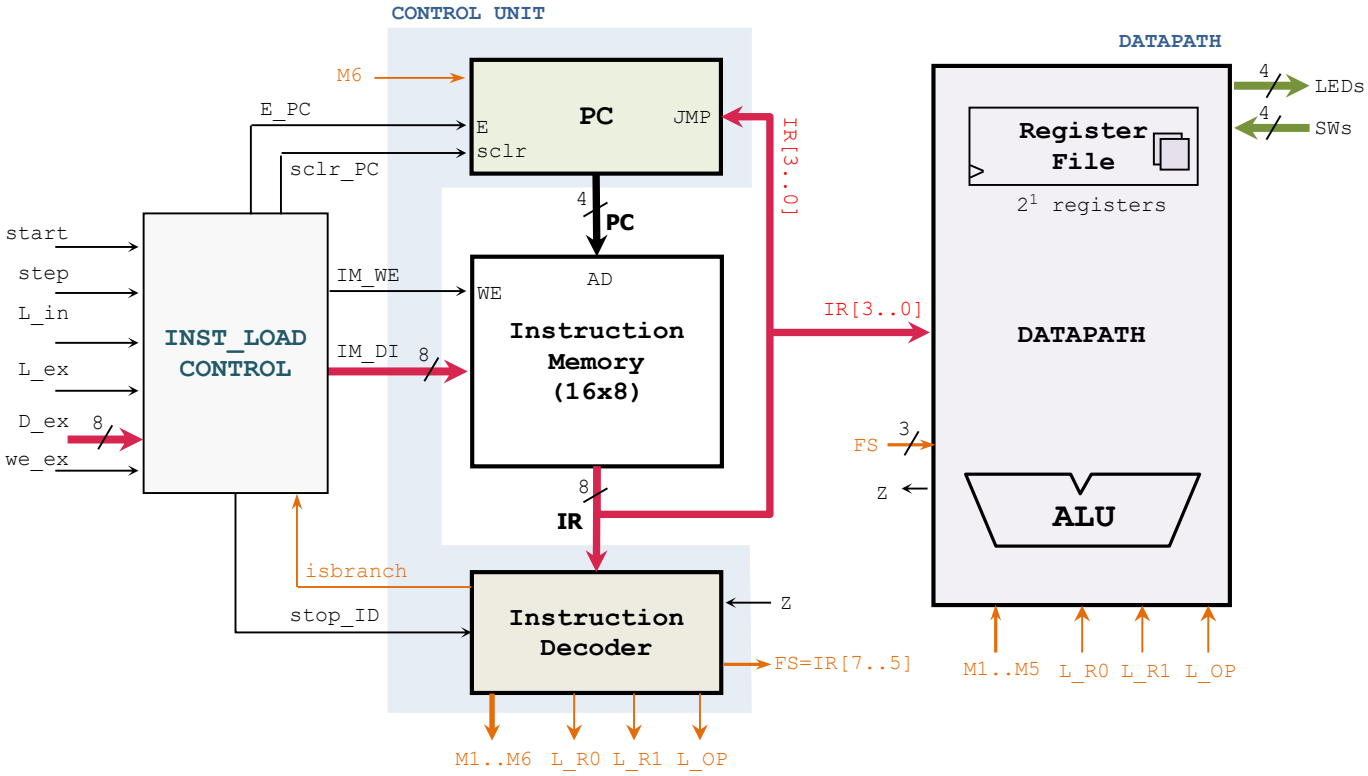
address	INSTRUCTION MEMORY
0000	00100010
0001	00110100
0010	11000000
0011	01100001
0100	01110001
0101	11110010
0110	00100001
0111	11100000
1000	00000000
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

```
* 2 to 13 ≡ 4 to 15


start: loadi R0,2          R0 ← 2
       loadi R1,4          R1 ← 4
loop:  out R0 → OUT: shows the count
       addi R0,1           R0 ← R0+1
       addi R1,1           R1 ← R1+1
       jnz R1, loop
       loadi R0,1          R0 ← 1
       jnz R0, start
```

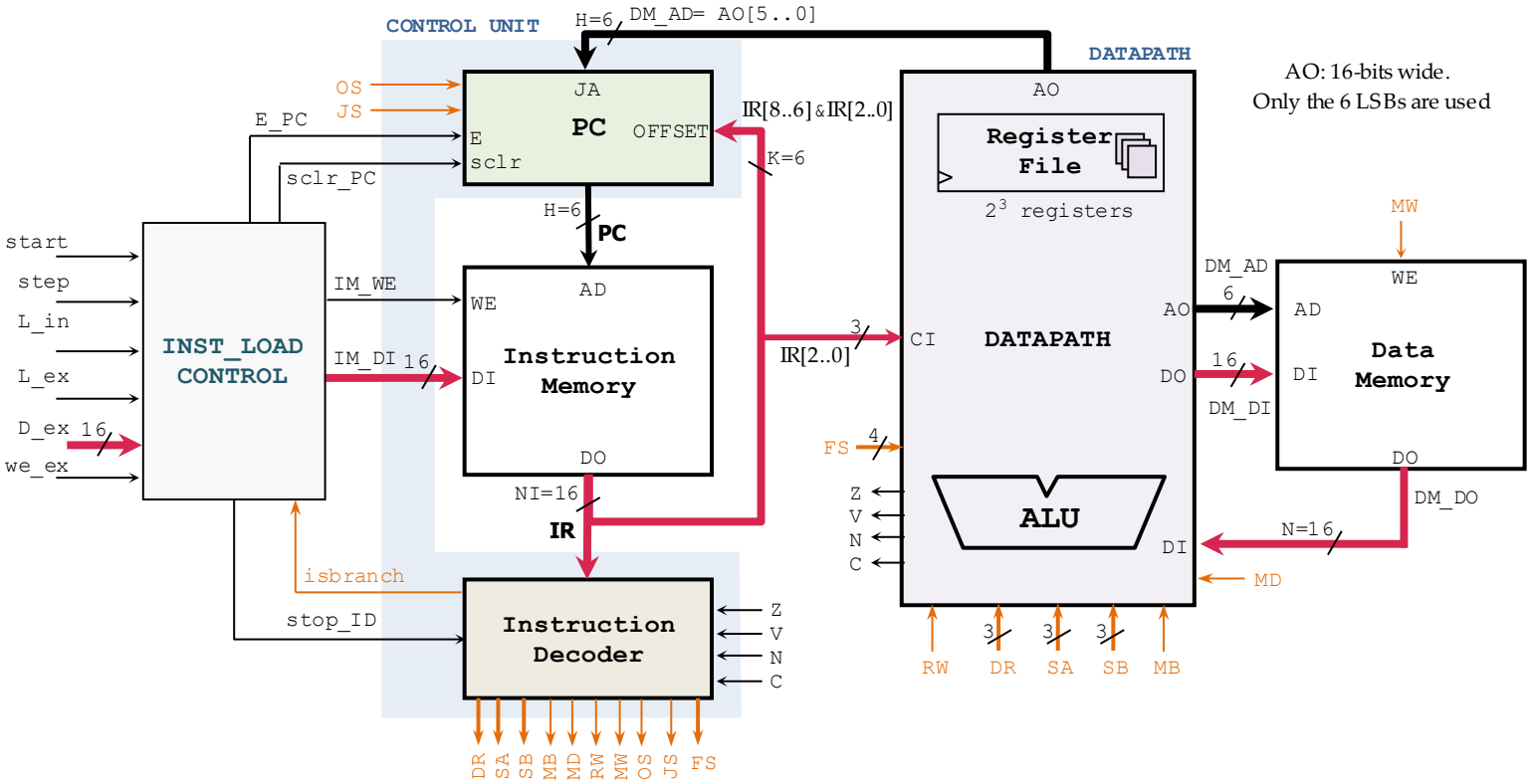
Microprocessor with Instruction Load Control for VBC Computer

- For hardware testing, we need to include an Instruction Load Control circuitry.
- The Instruction Load Control component can load instructions from a parallel input (one by one by asserting `we_ex`), or it can load a pre-defined set of instructions (by asserting `L_in`).
- The figure below shows this VBC computer (CPU and Instruction Memory) along with a circuit that controls the loading of instructions. Here, we specify the VBC computer using blocks along with their proper connections



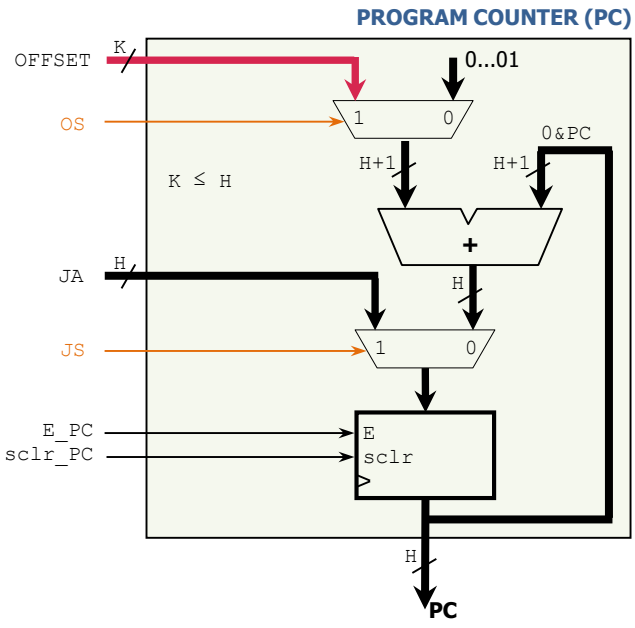
SINGLE-CYCLE HARDWIRED CONTROL – A SIMPLE COMPUTER

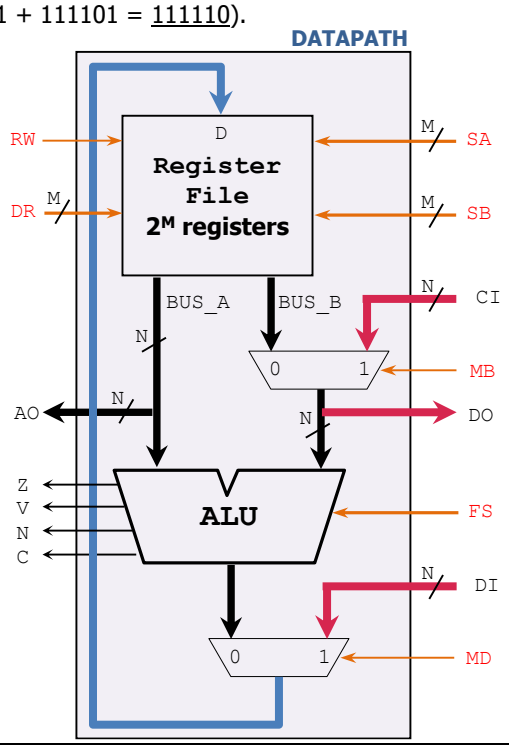
- Here, we provide a more formal description of a microprocessor (using the generic CPU model); the figure includes the Instruction Load Control component. Parameters: NI=16, N=16, K=H=6, M=3 (8 Registers). The Function Select (FS) of the ALU has 4 bits. The Constant Input (CI) of the Datapath has N=16 bits, where CI[2..0]=IR[2..0], and CI[15..3]="00…00"
- Instruction Load Control: It does not control loading of data into Data Memory, though it could be updated to handle that.



PROGRAM COUNTER (PC)

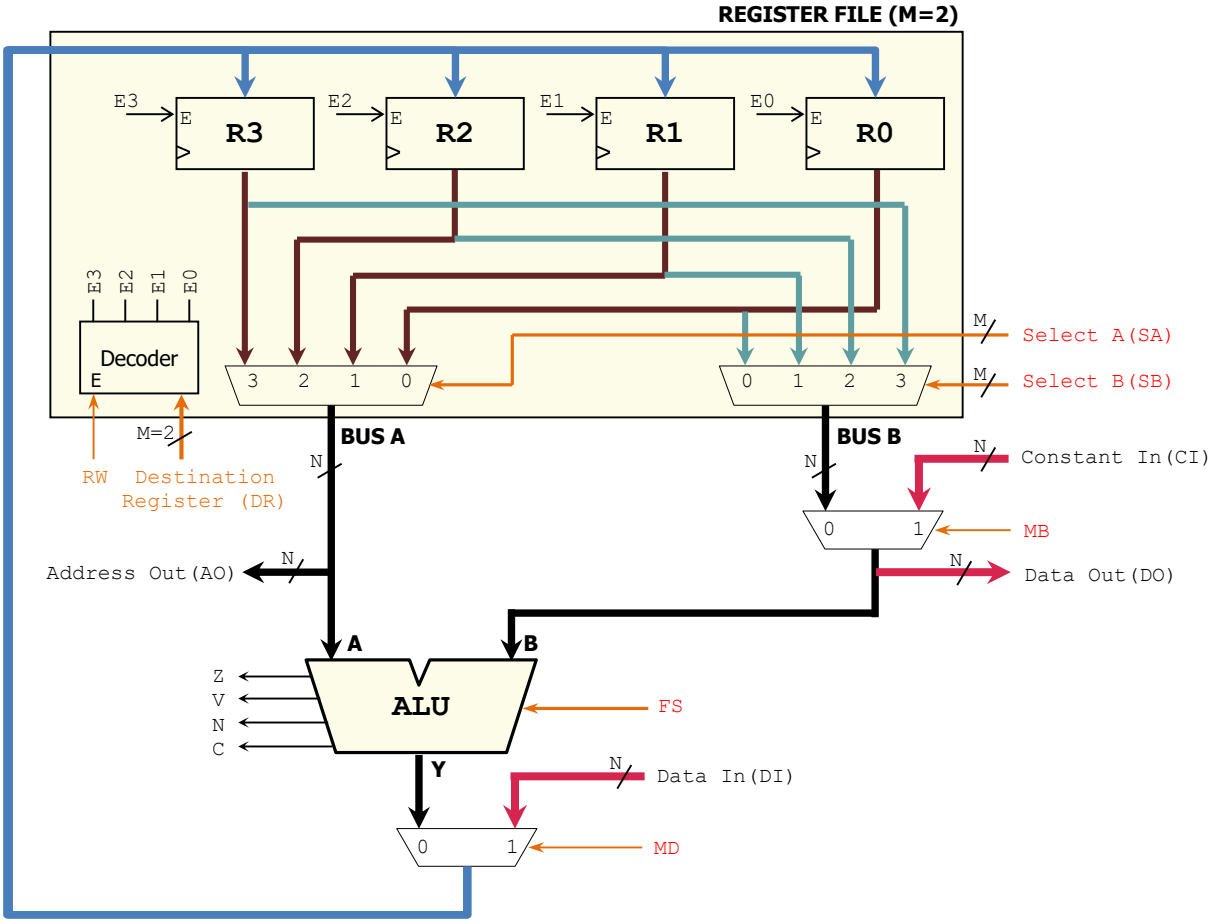
- This Generic Program Counter accepts a Jump Address (AO) and an Offset Address.
- Note that PC and JA are unsigned H-bit addresses, while OFFSET can be an unsigned or signed K-bit value ($K \le H$).
- In the figure, we use a signed offset. As a result, we zero extend PC and add it to the OFFSET resulting in H+1 bits. We only grab H bits and treat the result as unsigned. This means that if the result ends up being outside $[0, 2^H-1]$, we wraparound the result (e.g.: 111110 + 000011 = 1000001; 000001 - 000011 = 000001 + 111101 = 111110).





DATAPATH

- A generic datapath includes a Register File and an ALU (see previous figure). A Register File includes 2^M registers, so we need M bits to address all of these registers.
- **Register File**: The figure below depicts a Register File with M=2, resulting in 2^2=4 registers. Note how in this particular implementation, we use 2 data buses (Bus A and Bus B). Other implementations only use one Data Bus. We also include the connections to the ALU and to the Datapath inputs and outputs.



- **Arithmetic Logic Unit**: The FS has 4 bits, and the following table lists all the possible operations. The input Data (A, B) and output data (Y) are represented as signed numbers. Here, the flags Z, V, N, C are generated.

FS	Operation	Function	Flag bits	Unit
0000	Y <= A	Transfer A	N, Z	Arithmetic
0001	Y <= A + 1	Increment A	V, C, N, Z	
0010	Y <= A + B	Add A and B with cin=0	V, C, N, Z	
0011	Y <= A + B + 1	Add A and B with cin=1	V, C, N, Z	
0100	Y <= A – B – 1 = A + not(B)	Subtract B from A with bin=1	V, C, N, Z	
0101	Y <= A – B = A + not(B) + 1	Subtract B from A with bin=0	V, C, N, Z	
0110	Y <= A – 1	Decrement A	V, C, N, Z	
0111	Y <= B	Transfer B	N, Z	
1000	Y <= A OR B	Bit-wise OR	N, Z	Logic
1001	Y <= A AND B	Bit-wise AND	N, Z	
1010	Y <= A XOR B	Bit-wise XOR	N, Z	
1011	Y <= not A	Complement A	N, Z	
1100	Y <= not B	Complement B	N, Z	
1101	Y <= sr B	Right-shift B	N, Z	
1110	Y <= sl B	Left-shift B	N, Z	
1111	Y <= 0	Transfer 0	None affected	

✓ In this particular implementation, the carry out (C) from a previous operation is not an input to the ALU. Instead, we have to use a specific instruction that adds the carry in (or borrow in) to an operation when desired.

INSTRUCTION MEMORY AND DATA MEMORY

- Instruction Memory (IM): It stores up to 2^H=64 16-bit instructions. The Instruction Load Control Component allows for instructions to be loaded externally. The PC controls which instruction is to appear on the Instruction Register (IR).
- Data Memory (DM): It stores up to 2^H=64 16-bit data values. It allows us to load and store data values during program execution. Here, the Data Memory (DM) can only be loaded via the Datapath,

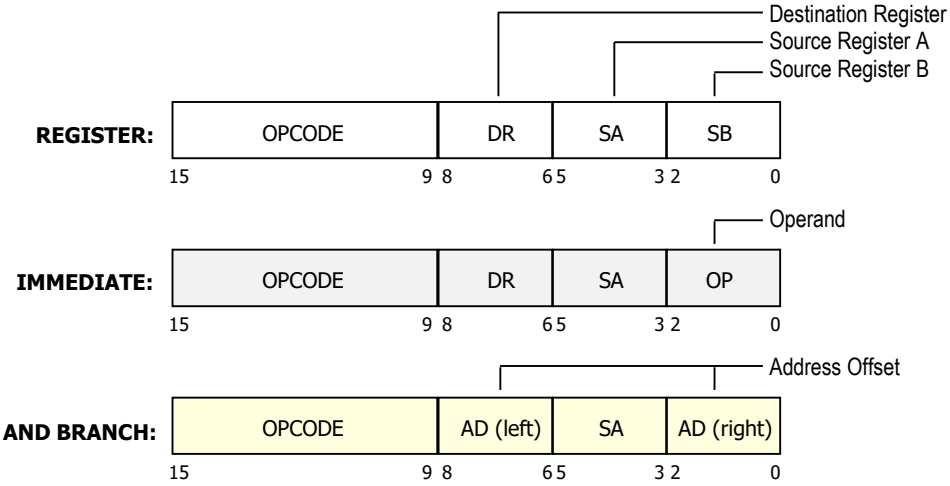
INSTRUCTION SET

- Instruction: Collection of bits that instructs the compute to perform a specific operation.
  - ✓ Each instruction specifies: i) an operation the system is to perform, ii) the registers or memory words where the operands are to be found and the result is to be placed, and/or iii) which instruction to execute next.
  - ✓ Instructions are usually stored in memory (RAM or ROM). To execute the instructions sequentially, we need the address in memory of the instruction to be executed. The address comes from the Program Counter (PC).
  - ✓ Executing an instruction means activating the necessary sequence of *microoperations* in the datapath (e.g.: add, subtract, load, shift) and elsewhere required to perform the operation specified by the instruction.
    - Operation: This is specified by an instruction in memory. The Control Unites decodes the instruction in order to perform the required microoperations for the execution of the instruction.
    - Microoperation: This is specified by the control bits generated by the Instruction Decoder (ID). The execution of a computer operation often requires a sequence of microperations, rather than a single microoperation.

- Instruction Set: Collection of instructions for a computer.
- Instruction Set Architecture (ISA): A thorough description of the instruction set. Simple ISAs have three major components: storage resources (IM, DM, Register File), instruction formats, and instruction specifications.
- Program: List of instructions that specifies the operations, the operands, and the sequence in which processing is to occur. It is where the user specifies the operations to be performed and their sequence.
  - ✓ The data processing performed by a computer can be altered by specifying a new program with different instructions or by specifying the same instructions with different data.
  - ✓ Instruction and Data can be stored in the same memory, in different memories, or they might appear to come from different memories.
  - ✓ The Control Unit reads an instruction from memory, decodes it, and executes the instruction by issuing a sequence of one or more microoperations (in single-cycle CPUs, we only perform microoperation per instruction).
  - ✓ The ability to execute a program from memory is the most important single property of a general-purpose computer.

Instruction Format

- The 16-bit instructions are generated by the Instruction Memory (IM) and written on the Instruction Register (IR). The instruction format might have different fields depending on the instruction type. Some microprocessors (like the VBC) only have one instruction type. In this particular implementation, we have 3 different instruction types:
  - ✓ Register: Opcode, 2 Source Registers (SA, SB), and a Destination Register (DR).
  - ✓ Immediate: Opcode, 1 Source Register (SA), a Destination Register (DR), and a 3-bit immediate operand (OP).
  - ✓ Jump and Branch: Opcode, Source Register, and 6-bit signed address offset: No register or memory contents are changed. Here, we only update the PC.
- The OPCODE specifies the operation to be executed, which must use data stored in the registers or in memory.



List of Instructions

- Each instruction is denoted with a symbolic notation: the OPCODE is given a mnemonic, and the additional instruction fields are denoted by literals. This symbolic notation (called Assembly Instruction), that represents the operation executed by the instruction, is then converted to the binary representation by a program called Assembler.

- The table provides the instruction specification, i.e., a description of the operation performed by each instruction, including the status bits affected by the instruction. We include a limited number of instructions; the designer can always add more instructions that are supported by the Datapath and Control Unit.

Instruction	Opcode	Mne-monic	Format	Description	PC	Status Bits
Move A	0000000	MOVA	RD, RA	R[DR] ← R[SA]	PC ← PC+1	N, Z
Increment	0000001	INC	RD, RA	R[DR] ← R[SA] + 1	PC ← PC+1	N, Z, C, V
Add	0000010	ADD	RD, RA, RB	R[DR] ← R[SA] + R[SB]	PC ← PC+1	N, Z, C, V
Subtract	0000101	SUB	RD, RA, RB	R[DR] ← R[SA] - R[SB]	PC ← PC+1	N, Z, C, V
Decrement	0000110	DEC	RD, RA	R[DR] ← R[SA] - 1	PC ← PC+1	N, Z, C, V
AND	0001000	AND	RD, RA, RB	R[DR] ← R[SA] ∧ R[SB]	PC ← PC+1	N, Z
OR	0001001	OR	RD, RA, RB	R[DR] ← R[SA] ∨ R[SB]	PC ← PC+1	N, Z
Exclusive OR	0001010	XOR	RD, RA, RB	R[DR] ← R[SA] ⊕ R[SB]	PC ← PC+1	N, Z
NOT	0001011	NOT	RD, RA	R[DR] ← not (R[SA])	PC ← PC+1	N, Z
Move B	0001100	MOVB	RD, RB	R[DR] ← R[SB]	PC ← PC+1	N, Z
Shift Right	0001101	SHR	RD, RB	R[DR] ← sr R[SB]	PC ← PC+1	N, Z
Shift Left	0001110	SHL	RD, RB	R[DR] ← sl R[SB]	PC ← PC+1	N, Z
Load Immediate	1001100	LDI	RD, OP	R[DR] ← OP	PC ← PC+1	N, Z
Add Immediate	1000010	ADI	RD, RA, OP	R[DR] ← R[SA] + OP	PC ← PC+1	N, Z
Load	0010000	LD	RD, RA	R[DR] ← M[R[SA]]	PC ← PC+1	
Store	0100000	ST	RA, RB	M[R[SA]] ← R[SB]	PC ← PC+1	
Branch on Zero	1100000	BRZ	RA, AD	If R[SA] ≠ 0 If R[SA] = 0	PC ← PC+1 PC ← PC+AD	N, Z
Brand on Negative	1100001	BRN	RA, AD	If R[SA] ≥ 0 If R[SA] < 0	PC ← PC+1 PC ← PC+AD	N, Z
Jump	1110000	JMP	RA		PC ← R[SA]	

- Other ISAs do not generate status bits when transfers on the Bus B (e.g. Move B) are occurring.
- Note that the branch instructions generate N, Z because they require Bus A to be transferred in order to evaluate R[SA] which might assert N or Z. The Jump instruction does not affect the status bits.
- Some considerations regarding the notation of the Instruction Description:
  - ✓ R[DR]: This refers to the register whose number is DR. Example: if DR=2 → R2.
  - ✓ M[R[SA]]: This refers to the memory address given by the value of the Register with number SA, e.g.: if SA=3 → M[R3].
- The following table shows an example with instructions in memory and a detailed description of them:

Address	Memory Contents	Other Fields	Assembly Instruction	Operation	Comments
011001	0000101001010011	DR:1, SA:2, SB:3	SUB R1,R2,R3	R1 ← R2 – R3	
100011	0100000000100101	SA:4, SB:5	ST R4,R5	M[R4] ← R5	DR unused
101101	1000010010111110	DR:2, SA:7, OP:6	ADI R2,R7,6	R2 ← R7 + 6	
110111	1100000101110100	AD:-20, SA:6	BRZ R6,-20	If R[SA] = 0: PC ← PC-20	-20=101100
111110	0010000101010000	DR:5, SA:2	LD R5,R2	R5 ← M[R2]	SB unused

Example:

- The following Assembly Program implements a counter from 2 to 13: 2,3,…, 13,2,3,…
  As we cannot use 11 as a 3-bit immediate operand, we first load 7 on R1 and then add 4. `* 2 to 13 ≡ 11 downto 0`
  We use `'---'` to indicate the values that are unused. This means we can assign any value to them.

Address	Instruction Memory	Assembly Program	
000000	1001100 011 --- 100	start: LDI R3,4	R3 ← 4
000001	1001100 000 --- 010	LDI R0,2	R0 ← 2
000010	1001100 001 --- 111	LDI R1,7	R1 ← 7
000011	1000010 001 001 100	ADI R1,R1,4	R1 ← R1+4 = 11
000100	1000010 000 000 001	loop: ADI R0,R0,1	R0 ← R0+1
000101	0000110 001 001 ---	DEC R1,R1	R1 ← R1-1
000110	1100000 111 001 011	BRZ R1,-5	
000111	1110000 --- 011 ---	JMP R3	
001000	0000000 000 000 000		R0 ← R0 (This is NOP operation)
...			

Example:

- The following Assembly Program stores numbers from 43 down to 29 in Data Memory (DM) on addresses 0 to 14.
  - ✓ After the instruction `ST R4,R6` is executed, the `R6` value appears on DM output. This is also true after the instruction `BRZ R4,-7` is executed. This is because these instructions cause `SA=4`, which in turn makes `AO=R4[5..0]`. And `R4[5..0]` is the DM address where the value of `R6` was stored.
  - ✓ At the instruction `JMP R2, DM_DO` shows the value at the address equal to `R2` (`SA=2` makes `AO = R2[5..0]`)

Address	Instruction Memory	Assembly Program	
000000	1001100 010 --- 101	start: LDI R2,5	R2 ← 5
000001	1001100 110 --- 111	LDI R6,7	R6 ← 7
000010	1000010 110 110 111	ADI R6,R6, 7	R6 ← 14
000011	0000000 100 110 ---	MOVA R4,R6	R4 ← 14
000100	0000010 110 100 110	ADD R6,R4,R6	R6 ← 28
000101	0000001 110 110 ---	loop: INC R6,R6	R6 ← R6+1
000110	0100000 --- 100 110	ST R4,R6	M[R4] ← R6
000111	1100000 111 100 001	BRZ R4,-7	If R4=0 ⇒ PC ← PC-7=0
001000	0000110 100 100 ---	DEC R4,R4	R4 ← R4-1
001001	1110000 --- 010 ---	JMP R2	PC ← R2=5
001010	0000000 000 000 000		(NOP operation)
...			

address	DM
000000	2B
000001	2A
000010	29
000011	28
000100	27
000101	26
000110	25
000111	24
001000	23
001001	22
001010	21
001011	20
001100	1F
001101	1E
001110	1D
⋮	

INSTRUCTION DECODER

- The inputs to this circuit are the Instruction Register (IR) and the V, C, N, Z flags. The outputs are all the control signals: DR, SA, SB, MB, MD, RW, MW, OS, JS, FS. In this implementation, the V, C, N, Z flags are only considered when branching.
- Also, if stop_ID=1, the following signals must be set to '0': register enables in the Datapath, the DM write enable, and the PC control signals OS and JS. This is useful to pause execution of a program (PC and Datapath are not updated).
- This is a combinational circuit. The I/O relationship depends on how each instruction is defined. We provide the output signals for some instructions:

	Instruction Register	V	C	N	Z	RW	DR	SA	SB	MB	MD	FS	MW	OS	JS
MOVA R1,R2	0000000001010---					1	001	010	---	-	0	0000	0	0	0
MOVA R7,R0	0000000111000---						111	000							
MOVB R0,R3	0001100000---011					1	000	---	011	0	0	0111	0	0	0
MOVB R6,R6	0001100110---110						110		110						
ADD R3,R2,R1	0000010011010001					1	011	010	001	0	0	0010	0	0	0
ADD R6,R0,R0	0000010110000000						110	000	000						
XOR R6,R1,R3	0001010110001011					1	110	001	011	0	0	1010	0	0	0
XOR R5,R4,R5	0001010101100101						101	100	101						
LDI R7,3	1001100111---011					1	111	---	---	1	0	0111	0	0	0
LDI R5,4	1001100101---100						101		---						
ADI R0,R1,7	1000010000001111					1	000	001	---	1	0	0010	0	0	0
ADI R2,R6,3	1000010010110011						010	110	---						
LD R3,R7	0010000011111---					1	011	111	---	-	1	1111	0	0	0
ST R1,R5	0100000---001101					0	---	001	101	0	-	1111	1	0	0
BRN R4,-5	1100001111100011			0		0	---	100	---	-	-	0000	0	0	0
				1										1	0
BRZ R3,12	1100000001011100				0	0	---	011	---	-	-	0000	0	0	0
					1									1	0
JMP R5	1110000---101---					0	---	101	---	-	-	1111	0	-	1

- Branch instructions (BRN, BRZ): These instructions might affect the N and Z bits. Depending on how they affect these flag bits, we either branch or increase the value of the PC.
- JMP, LD, ST: They use FS=1111 since in this case the V, C, N, Z flags are unaffected.

Memory latency (IM, DM)

- Memory implemented as an array of registers: Writing: Data takes 1 clock cycle to be written. Reading: Output data output appears as soon as address is ready (assuming no extra output register).
- Memory implemented using BRAMs (assuming no extra output register): Writing: Data takes 1 clock cycle to be written. Reading: Output data takes 1 clock cycle to appear when address is presented (this is, address is read on the clock edge).
- Other memory technologies (SRAMs, DDRRAMs, etc.): Writing/Reading: It might take many cycles for data to be written or to appear on the output.

Single-Cycle Computer Shortcomings:

- ALU operations that might require more than one cycle to execute (e.g. multiplication, division) cannot be executed, or they would require a large combinational delay.
- Lower limit on the clock period based on a long worst-case delay path. Pipelining of the datapath is required to reduce the combinational delay between registers. This requires multiple-cycle control.

MULTI-CYCLE HARDWIRED CONTROL – PICOBLAZE™ EMBEDDED MICROPROCESSOR

- This is a commercial example of an 8-bit microprocessor developed by Xilinx® (datasheet). We present an adapted version here (hardware details were inferred based on the datasheet specification). This microprocessor design is meant to be instantiated into a large design that combines microprocessor and a digital design (in an FPGA).
- The architecture depicts known components as well as new components and features (e.g.: Stack and Interrupt handling).
- Each instruction takes 2 cycles to complete.

- I/O interface: It consists of 5 signals.

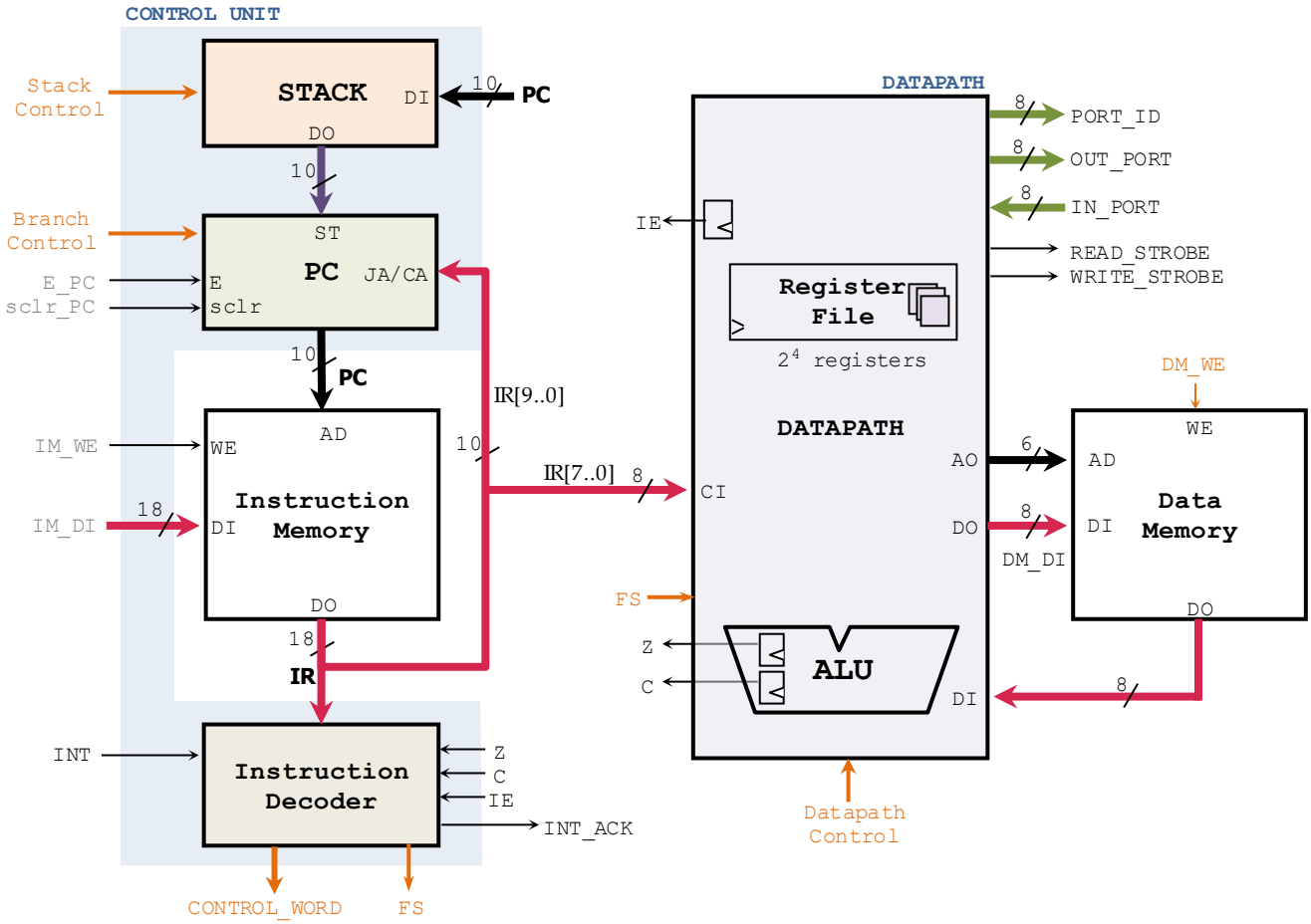
- Register File: 16 8-bit registers: s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sA, sB, sC, sD, sE, sF.
- ALU: 8-bits. It supports:
  - ✓ addition/subtraction with or without carry, bit-wise AND, OR, XOR
  - ✓ arithmetic compare and bit-wise test operations
  - ✓ shift and rotate operations
- Flags: C, Z, IE. They are stored in flip flops. This is different from the previous examples.

- Instruction Memory (IM): 1024 18-bit words. To optimize resources on an FPGA, this memory is implemented with BRAMs instead of registers. So, there is a **one-cycle delay** when reading data.
  - ✓ File for BRAM-based memory implementation (Artix-7 FPGA or 7-series PL): `in_RAMgen.vhd`)

- Data Memory (DM): 64 8-bit words. It is called a ScratchPad memory. No delay: built out of decoder, register, and a MUX.

- Program Counter (PC): 10-bit. It supports up to 1024 instructions (`0x000` to `0x3FF`). When the PC reaches `0x3FF`, it rolls over to location `0x000`. Computed jump instructions (like offset) are not supported, i.e., the Datapath does not control the PC. The Stack Data might or might not be incremented by 1 by the Program Counter.

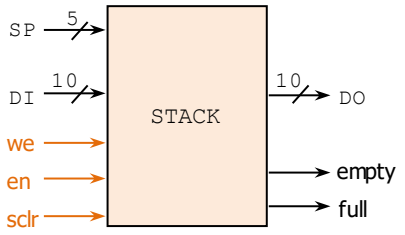
- Call/Return Stack: 31 locations (10-bit words). This allows the processor to handle nested subroutines.
- Interrupts: 5 cycles to respond and start ISR. An interrupt enable flag (IE) is required.



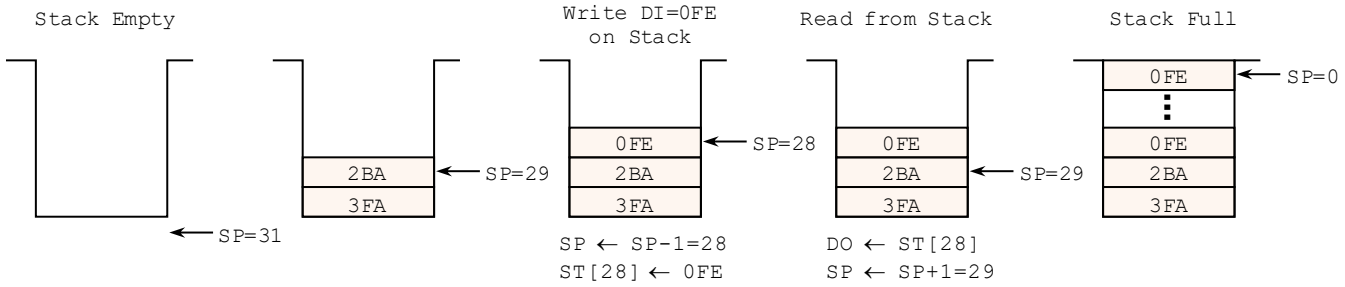
CALL/RETURN STACK

- This LIFO structure allows for the implementation of functions (including nested functions) as well as handling interrupts.
- It stores up to 31 10-bit instruction addresses, enabling nested function calls up to 31 levels deep. Address: 5-bit pointer called Stack Pointer (SP) or Top of Stack (TOS). DI: input data. DO: output data.
  - ✓ At power-up: SP ← 31 (Stack is empty)
  - ✓ If en = '1':
    - If we = '1': SP ← SP-1 ST[SP] ← DI
    - If we = '0': DO ← ST[SP] SP ← SP+1

    else: DO ← ST[SP]
  - ✓ empty flag = 1 if SP = 31, else 0 full flag = 1 if SP = 0, else 0



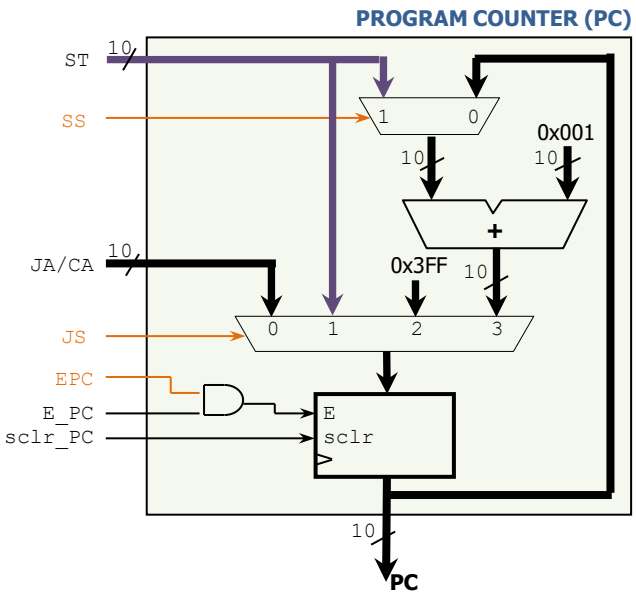
- SP is 5-bits wide as SP ∈ [0,31]. Note that SP=31 means Stack is empty. Thus, there are only 31 addresses (0 to 30) where we can write data. ST[SP] denotes the contents of the Stack at address SP.
- The figure shows different stages of a Stack. When data is written onto the Stack, we say that we push a value onto it. When data is read from the Stack, we say that we pop a value from it.

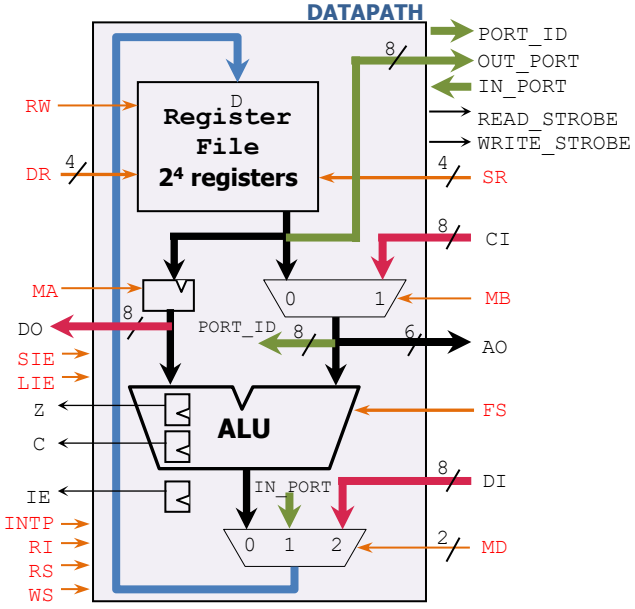


- Writing on the Stack (SP ← SP-1 followed by ST[SP] ← DI): To minimize delays, these operations are usually executed at the same time. This is, the hardware precomputes SP-1, and it executes ST[SP-1] ← DI and SP ←SP-1 simultaneously.
- Reading from Stack: To minimize delays, there is usually no latency on DO, i.e., DO is already showing the Top of the Stack.
- Subroutines Calls and Interrupt Event Handling:
  - ✓ Call to Subroutine (or to ISR): To save PC on Stack, the Instruction Decoder issues `we=en=1, sclr=0`.
  - ✓ Return from Subroutine (or from ISR): To restore PC from Stack, the Instruction Decoder issues `we=0, en=1, sclr=0`.
- PicoBlaze: The Call/Return Stack is implemented as a cyclic buffer. When the Stack is Full, it overwrites the oldest value.

PROGRAM COUNTER (PC)

- This Program Counter accepts a Jump/Call Address (`JA/CA`) and an address from the Stack Pointer (`ST`).
- The value of PC is determined by the control signals coming from the Instruction Decoder:
  - ✓ Subroutines: On a Call, the PC value (subroutine address) is given by `JA/CA`. On return, PC is loaded with `ST+1`.
  - ✓ Interrupts: PC gets the value `0x3FF`. On return from interrupt, PC gets the value `ST`.
- Due to the Instruction Memory (`IM`) one-cycle reading latency, a PC value must be available a cycle before its instruction appears on `IR`. Hence, a pBlaze instruction needs two cycles: we must wait a cycle after PC is updated to get an instruction from `IM`. PC lasts two cycles and it is updated by `EPC` (which is gated with the external `E_PC` signal) generated by the `ID`.





DATAPATH

- This datapath (see previous figure) includes: i) a Register File with 2^4 8-bit registers, ii) an ALU that stores the flags C and Z on flip flops, and iii) an I/O interface.
- The Datapath executes the microoperations required by an instruction based on the Control signals received from the Instruction Decoder (ID):
  - ✓ IE: We can set this flag bit to '1' or '0' at any time (using the signals `SIE` and `LIE`).
  - ✓ Interrupt Handling: On an interrupt event, the datapath stores Z an C onto buffers (ZI and CI) and clears IE. On return from interrupt, Z and C are restored (with the ZI and CI values), and IE can be set to either '1' or '0'.
  - ✓ I/O interface: `READ_STROBE` and `WRITE_STROBE` come from the Instruction Decoder (`RS` and `WS`).

- **Register File**: The architecture resembles that of the 'Simple Computer', except that there is only one output data bus.

- **Arithmetic Logic Unit (ALU)**: The FS has 5 bits, and the following table lists all the possible operations. The input Data (A, B) and output data (Y) can be thought of unsigned or signed integers. The exception is the subtraction operation where A and B are unsigned integers. Here, the flags Z, C are generated.

FS	Operation	Function	Flag bits		Unit
00000	`Y <= A`	Transfer A		No flags affected	Arithmetic
00001	`Y <= A + B`	Add A and B	C, Z		
00010	`Y <= A + B + c`	Add A and B with C=cin	C, Z		
00100	`Y <= A - B`	Subtract B from A	C, Z	Unsigned subtraction, C represents borrow out	
00101	`Y <= A - B - c`	Subtract B from A with C=bin	C, Z		
00111	`Y <= A AND B`	Bit-wise AND	C, Z	C ← 0	Logic
01000	`Y <= A AND B, tst`	Bit-wise AND, C different	C, Z	C ← Y(7)⊕Y(6)… ⊕Y(0)	
01001	`Y <= A OR B`	Bit-wise OR	C, Z	C ← 0	
01010	`Y <= A XOR B`	Bit-wise XOR	C, Z		
01101	`Y <= sL A 0`	Left-shift A, din = 0	C, Z	C ← A(7)	
01110	`Y <= sL A 1`	Left-shift A, din = 1	C, Z		
01111	`Y <= sL A A0`	Left-shift A, din = A(0)	C, Z		
10000	`Y <= sL A c`	Left-shift A, din = C	C, Z		
10001	`Y <= sR A 0`	Right-shift A, din = 0	C, Z	C ← A(0)	
10010	`Y <= sR A 1`	Right-shift A, din = 1	C, Z		
10011	`Y <= sR A A7`	Right-shift A, din = A(7)	C, Z		
10100	`Y <= sR A c`	Right-shift A, din = C	C, Z		
10101	`Y <= rL A`	Rotate left A	C, Z	C ← A(7)	
10110	`Y <= rR A`	Rotate right A	C, Z	C ← A(0)	

  - ✓ Here, the flags C and Z are stored in FFs each time the ALU executes an operation that changes C and Z. We can use the C flag in our operations. If we need to use C, we grab the value from the flip flop.
  - ✓ Note that unlike the 'Simple Computer' model, the C flag (carry out/borrow out) is an input to the ALU. We can then use a specific instruction that uses the value of this C flag from the flip flop. This allows us to implement, for example, multi-precision addition and subtraction.
  - ✓ To handle interrupts, on an interrupt event we must store C and Z onto other ffs CI and ZI. On return from interrupt, we restore the values of C and Z from CI and ZI. The flip flops CI and ZI are part of this ALU.

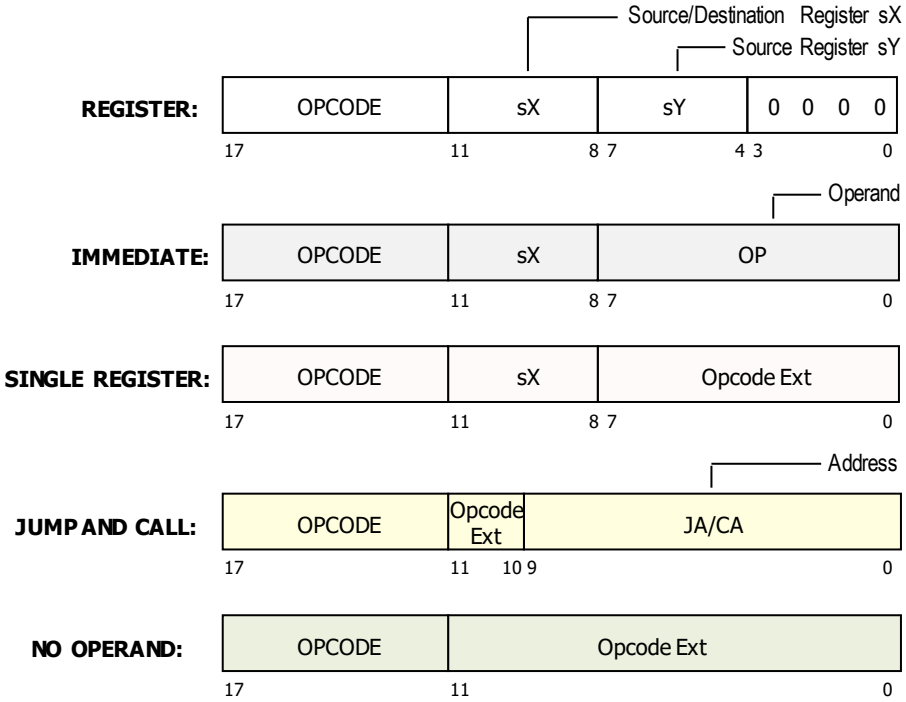
- All instructions require two cycles. However, note that in this architecture, some operations (e.g.: compute the result of two registers, compute the result of a register and a constant, reading/writing from external ports) do require 2 cycles to update the results. On the other hand, other instructions (e.g.: reading and writing from Data Memory, clearing IE) take only 1 cycle. As a result, in this multi-cycle processor, we will see instructions that either require 1 or 2 microoperations to update registers, memory contents, and flags.
- This multi-cycle microprocessor has a uniform number of cycles (2) per instructions. Note that other processor can take different number of cycles depending on the instruction, thereby making the control mechanism more complex.

INSTRUCTION SET

- Note that it is common practice to specify (design) the Instruction Set first and then build the Datapath based on the instructions that need to be supported.

Instruction format:

- 18-bit instruction. The instruction format has different fields depending on the instruction type. PicoBlaze has 5 different instruction types. Note that the Destination Register is given by sX.
  - ✓ Register: Opcode, 2 Source Registers (sX, sY).
  - ✓ Immediate: Opcode, 1 Source Register (sX), 8-bit immediate operand (OP).
  - ✓ Single Register: Opcode, 1 Source Register (sX), 8-bit Opcode extension that further specifies the operation.
  - ✓ Jump and Call: Opcode, 2-bit Opcode extension, 10-bit immediate operand (JA/CA).
  - ✓ No Operand: Opcode, 12-bit Opcode extension.



List of Instructions

- The table provides the description of the operation performed by each instruction, as well as the status bits affected by the instruction. We provide the instructions in Assembly instruction format (mnemonic followed by literals).
  - ✓ Constants Operands: kk = OP[7..0], ss = OP[5..0] (OP[7..6]=00), aaa = JA/CA[9..0].
  - ✓ Addition: sX, sY, kk can be treated as unsigned or signed integers.
  - ✓ Subtraction (unsigned): sX, sY, kk are treated as unsigned integers. C is interpreted as the borrow out (or borrow in).
- Status Bits (C, Z): They are stored in FFs. An instruction read these bits when they are part of the operation. These bits can be updated after the instruction is executed. Z ← 1 if the result of the operation is 0. C ← 1 depending on the instruction.
- IN_PORT, OUT_PORT: I/O port names. PORT_ID: identifier or port address for an associated I/O operation.
- SP = TOS (top of the stack). ST[SP]: contents of the stack at address SP.
- Interrupts: CI, ZI: extra buffers to store C and Z when an interrupt hits so that we can restore them after the interrupt event. IE: Interrupt enable flag (also considered a status bit).
  - ✓ On an interrupt event, the following occurs: CI ← C, ZI ←Z, IE ← 0, ST[SP] ← PC, SP ← SP-1, PC ← 3FF.

PROGRAMMING MODEL

- From the point of view of the programmer, PicoBlaze contains:
  - ✓ 16 8-bit registers (s0-sF).
  - ✓ 64-byte Data Memory (DM).
  - ✓ 3 Status flags (C, Z, IE)
  - ✓ Program Counter (PC): This 10-bit pointer handles a 1024-word Instruction Memory (IM).
  - ✓ Stack Pointer (SP): this 5-bit pointer handles a 31-word Call/Return Stack.
- After an instruction is executed, the contents of these components are altered explicitly or implicitly.

Assembly Instruction	opcode	Description	PC, SP	Status bits
LOAD sX, sY	000001	sX ← sY	PC ← PC+1	
ADD sX, sY	011001	sX ← sX + sY	PC ← PC+1	C, Z
ADDCY sX, sY	011011	sX ← sX + sY + C	PC ← PC+1	C, Z
SUB sX, sY	011101	sX ← sX - sY * unsigned subtraction	PC ← PC+1	C, Z
SUBCY sX, sY	011111	sX ← sX - sY - C * unsigned subtraction	PC ← PC+1	C, Z
COMPARE sX, sY	010101	sX – sY, * unsigned subtraction, get C, Z	PC ← PC+1	C, Z
AND sX, sY	001011	sX ← sX ∧ sY, C ← 0	PC ← PC+1	C, Z
OR sX, sY	001101	sX ← sX ∨ sY, C ← 0	PC ← PC+1	C, Z
XOR sX, sY	001111	sX ← sX ⊕ sY, C ← 0	PC ← PC+1	C, Z
TEST sX, sY	010011	T = sX ∧ sY, C ← T(7)⊕T(6)...⊕T(0)	PC ← PC+1	C, Z
LOAD sX, kk	000000	sX ← kk	PC ← PC+1	
ADD sX, kk	011000	sX ← sX + kk	PC ← PC+1	C, Z
ADDCY sX, kk	011010	sX ← sX + sY + C	PC ← PC+1	C, Z
SUB sX, kk	011100	sX ← sX - kk * unsigned subtraction	PC ← PC+1	C, Z
SUBCY sX, kk	011110	sX ← sX – kk - C * unsigned subtraction	PC ← PC+1	C, Z
COMPARE sX, kk	010100	sX – kk, * unsigned subtraction, get C, Z	PC ← PC+1	C, Z
AND sX, kk	001010	sX ← sX ∧ kk, C ← 0	PC ← PC+1	C, Z
OR sX, kk	001100	sX ← sX ∨ kk, C ← 0	PC ← PC+1	C, Z
XOR sX, kk	001110	sX ← sX ⊕ kk, C ← 0	PC ← PC+1	C, Z
TEST sX, kk	010010	T = sX ∧ kk, C ← T(7)⊕T(6)...⊕T(0)	PC ← PC+1	C, Z
FETCH sX, (sY)	000111	sX ← M[sY] * only sY[5..0]	PC ← PC+1	
STORE sX, (sY)	101111	M[sY] ← sX * only sY[5..0]	PC ← PC+1	
INPUT sX, (sY)	000101	sX ← IN_PORT, PORT_ID ← sY	PC ← PC+1	
OUTPUT sX, (sY)	101101	OUT_PORT ← sX, PORT_ID ← sY	PC ← PC+1	
FETCH sX, ss	000110	sX ← M[ss] * ss: 6 bits	PC ← PC+1	
STORE sX, ss	101110	M[ss] ← sX * ss: 6 bits	PC ← PC+1	
INPUT sX, kk	000100	sX ← IN_PORT, PORT_ID ← kk	PC ← PC+1	
OUTPUT sX, kk	101100	OUT_PORT ← sX, PORT_ID ← kk	PC ← PC+1	
RL sX	100000	sX ← sX[6..0]&sX[7], C ← sX[7]	PC ← PC+1	C, Z
RR sX		sX ← sX[0]&sX[7..1], C ← sX[0]	PC ← PC+1	C, Z
SL0 sX		sX ← sX[6..0]&'0', C ← sX[7]	PC ← PC+1	C, Z
SL1 sX		sX ← sX[6..0]&'1', C ← sX[7]	PC ← PC+1	C, Z
SLA sX		sX ← sX[6..0]&sX[0], C ← sX[7]	PC ← PC+1	C, Z
SLX sX		sX ← sX[6..0]&C, C ← sX[7]	PC ← PC+1	C, Z
SR0 sX		sX ← '0'&sX[7..1], C ← sX[0]	PC ← PC+1	C, Z
SR1 sX		sX ← '1'&sX[7..1], C ← sX[0]	PC ← PC+1	C, Z
SRA sX		sX ← C&sX[7..1], C ← sX[0]	PC ← PC+1	C, Z
SRX sX		sX ← sX[7]&sX[7..1], C ← sX[0]	PC ← PC+1	C, Z
CALL aaa	110000	Go to subroutine in address aaa	SP ← SP-1, ST[SP] ← PC PC ← aaa	
CALL C, aaa	110001	Go to subroutine in address aaa if C=1	If Condition is met: SP ← SP-1, ST[SP] ← PC PC ← aaa else PC ← PC+1	
CALL NC, aaa		Go to subroutine in address aaa if C=0		
CALL Z, aaa		Go to subroutine in address aaa if Z=1		
CALL NZ, aaa		Go to subroutine in address aaa if Z=0		
JUMP aaa	110100	Go to instruction in address aaa	PC ← aaa	
JUMP C, aaa	110101	Go to instruction in address aaa if C=1	If Condition is met: PC ← aaa else PC ← PC+1	
JUMP NC, aaa		Go to instruction in address aaa if C=0		
JUMP Z, aaa		Go to instruction in address aaa if Z=1		
JUMP NZ, aaa		Go to instruction in address aaa if Z=0		
RETURN	101010	Return from Subroutine	PC ← ST[SP]+1, SP ← SP+1	
RETURN C	101011	Return from Subroutine if C=1	If Condition is met: PC ← ST[SP]+1, SP ← SP+1 else PC ← PC+1	
RETURN NC		Return from Subroutine if C=0		
RETURN Z		Return from Subroutine if Z=1		
RETURN NZ		Return from Subroutine if Z=0		
DISABLE INTERRUPT	111100	IE ← 0	PC ← PC+1	IE
ENABLE INTERRUPT		IE ← 1	PC ← PC+1	IE
RETURNI DISABLE	111000	Return from ISR, IE ← 0, C ← CI, Z ← ZI	PC ← ST[SP], SP ← SP+1	C, Z, IE
RETURNI ENABLE		Return from ISR, IE ← 1, C ← CI, Z ← ZI	PC ← ST[SP], SP ← SP+1	C, Z, IE

PROGRAM FLOW CONTROL

- During program execution, the PC provides the address of the instruction being executed.
  - ✓ Instructions are usually executed sequentially. Hence for most instructions: PC ← PC + 1.
  - ✓ Other instructions can alter program execution by loading a value onto PC:
    - Jump and branch instructions. We can go to any other instruction (via an absolute or relative address) in the program. Among other things, this allows for the implementation of loops.
    - Some processors include call and return instructions in order to handle subroutines and interrupt events.

    These instructions can be conditional or unconditional usually based on the ALU flags.

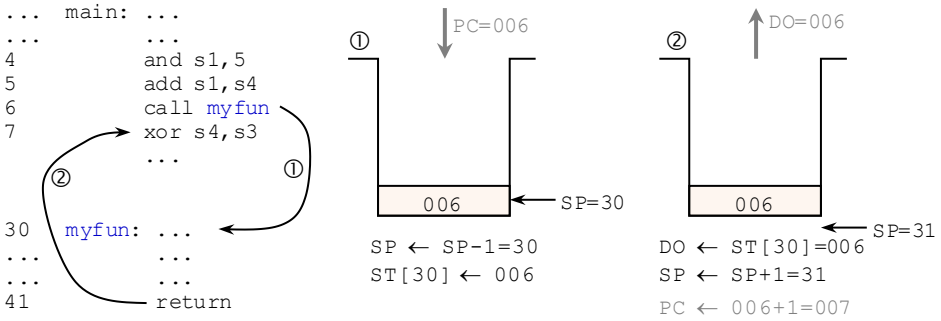
- To alter program execution, the PicoBlaze processor includes a variety of instructions:
  - ✓ JUMP instructions: We can go to a specific instruction (specified by an absolute address).
  - ✓ To handle subroutines, it has CALL and RETURN instructions.
  - ✓ To handle interrupt events, it has RETURNI instructions.

  The JUMP and CALL instructions can be executed unconditionally or conditionally based on the C and Z flags.

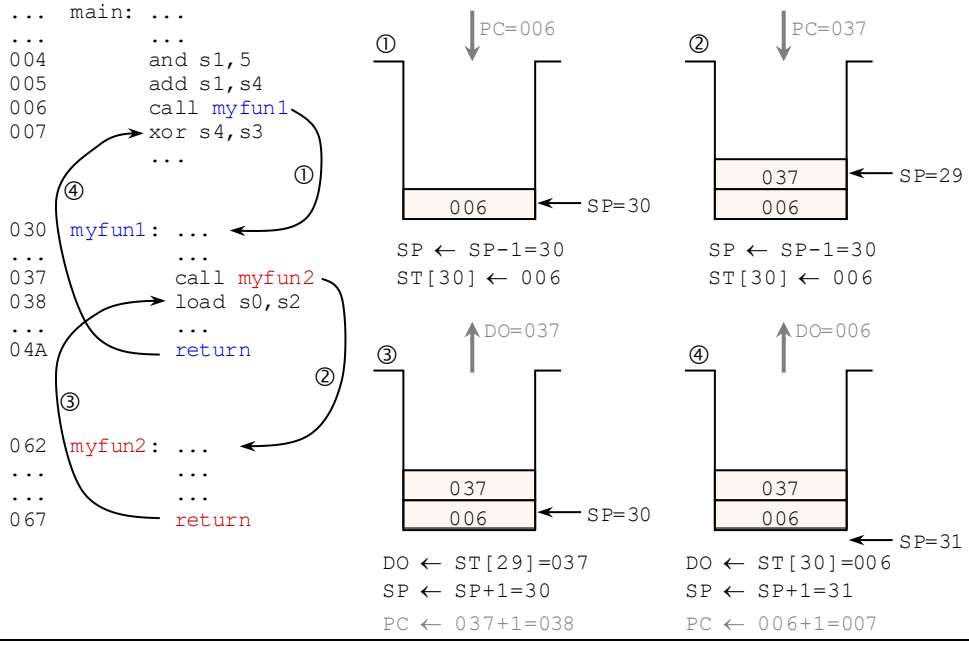
SUBROUTINES (PicoBlaze)

- The CALL and RETURN instructions can implement a subroutine. Here, we need to interact with the Call/Return Stack.
- A subroutine call is started by the CALL instruction. The process goes as follows:
  - ✓ The PC value is pushed on the Top of the Call/Return Stack: SP ← SP-1, ST[SP] ← PC
  - ✓ The CALL instruction jumps to the start of a subroutine (aaa address). PC ← aaa
  - ✓ Instructions in the subroutine are then executed until a RETURN instruction is reached. At that moment, the stored PC value is popped from Top of the Call/Return Stack, incremented by 1, and loaded onto PC: PC ← ST[SP]+1, SP ← SP+1
  - ✓ The program returns to the instruction immediately after the original CALL instruction.
- Note that only PC is saved. The registers (s0-sF) and flags (C, Z) are not saved. Consider this when using subroutines.
- The following examples use unconditional CALL and RETURN instructions. This can be easily modified to allow conditional CALL and RETURN instructions based on C and Z.

  - ✓ Example: A subroutine call: Program Flow and Stack state.



  - ✓ Example: Nested subroutine calls (2 levels): Program Flow and Stack state. Note how the Stack structure allows to save (and restore) the PC values in the right order.

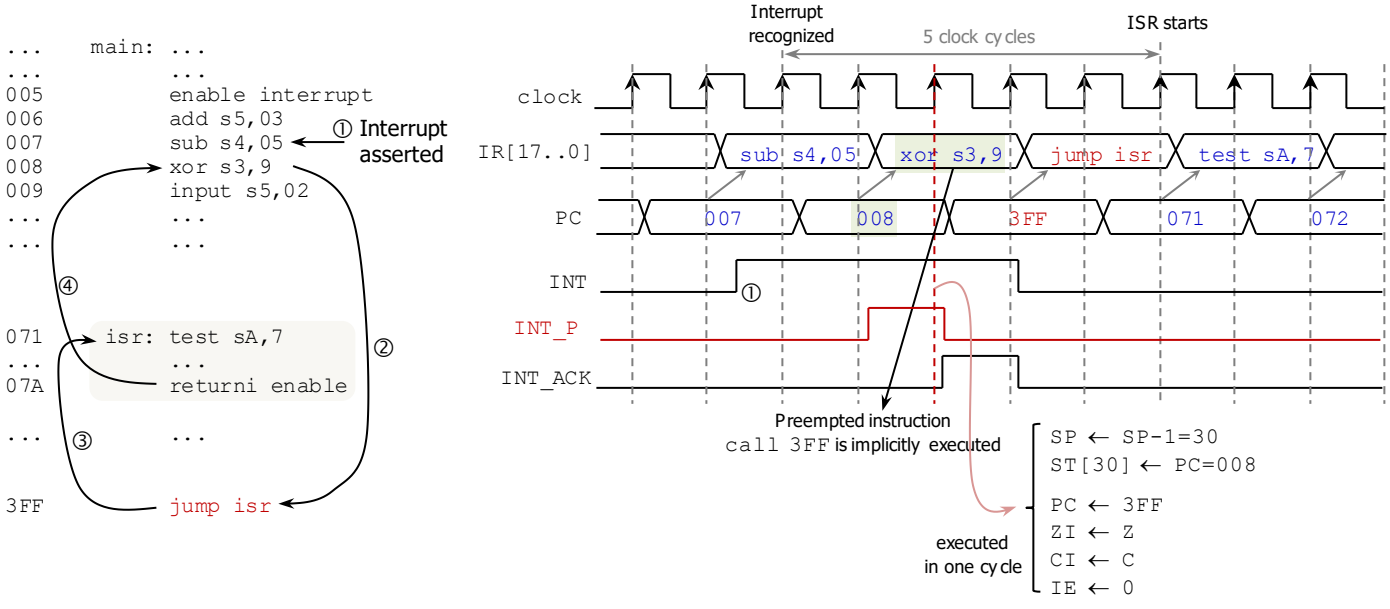


INTERRUPTS

- This is another mechanism to alter program execution. It is not initiated by an instruction, but rather by an external (to the microprocessor) request. When an interrupt event hits, the CPU stops normal program execution and perform some service (called Interrupt Service Routine) associated to the interrupt event, then returns to normal program execution.
- PicoBlaze™ provides a single interrupt signal `INT`. We can enable or disable this signal via the IE (interrupt enable) flag:
  - ✓ To enable interrupts, we use the ENABLE INTERRUPT instruction (IE ← 1)
  - ✓ To disable interrupts, we use the DISABLE INTERRUPT instruction (IE ← 0)
- Flow of an interrupt event:
  - ✓ By default, after reset (input signal to the PicoBlaze), the `INT` input is disabled. We need to enable interrupts first.
  - ✓ Once enabled, the `INT` signal must be asserted for at least two cycles in order to be recognized as an interrupt event.
  - ✓ An interrupt forces PicoBlaze to (implicitly) execute the CALL 3FF instruction immediately after completing the instruction being executed. The CALL 3FF instruction is a subroutine call to the last Instruction Memory (IM) location (`0x3FF`). During the implicit execution of the CALL 3FF instruction, the following occurs:
    - Further interrupts are disabled: IE ← 0
    - Flags Z and C are saved in buffers: CI ← C, ZI ←Z
    - The PC value pointing to the instruction pre-empted by CALL 3FF is saved on the Stack: SP ← SP-1, ST[SP] ← PC
    - The PC value is updated: PC ← 3FF (updated at the same time its current value is placed onto the Stack).
  - ✓ At `0x3FF`, there is typically a JUMP instruction to a subroutine called the Interrupt Service Routine (ISR).
  - ✓ The RETURNI DISABLE/ENABLE instruction ensures the end of ISR. When it is executed, the PC value as well as the Z and C values are restored. We can exit interrupt process with either enabling or disabling IE. When the interrupt process is finished, we need to execute the pre-empted instruction.

- The figure shows the flow of an interrupt event for a program example.
  - ✓ Note that the PC value last two cycles and it is available one cycle before the instruction. This is because pBlaze uses an Instruction Memory that takes a cycle to generate output data (BRAM). Also, PC lasts for two clock cycles.
  - ✓ `INT` signal: If asserted, it is recognized on the first immediate clock edge. However, it must be asserted for at least two clock cycles so that a FSM (inside the Instruction Decoder) detects `INT` on two clock edges and then generates a one-cycle pulse `INT_P` (during the first cycle of CALL 3FF). This pulse is used to save Z, C, and PC, as well as to update PC (PC ← 3FF) and clear IE. So, even though CALL 3FF does not appear on `IR`, it is implicitly executed via `INT_P`.
  - ✓ `INT_ACK`: This is a delayed version of `INT_P`. `INT_ACK` is asserted on the second cycle of the CALL 3FF instruction to indicate that the interrupt was recognized. `INT_ACK` can be used by an external interface to clear external interrupts.
  - ✓ The RETURNI instruction restores PC (PC ← ST[SP], SP ← SP+1). Unlike the RETURN instruction, the restored PC value from the Stack is not increased by 1. This is because the `INT_P` pulse saved the PC of the pre-empted instruction XOR s3,9 (`0x008`), which now needs to be executed.
  - ✓ Note that interrupt processing takes 5 cycles, this is from the moment the interrupt is recognized until the ISR starts.

- The starting address of the ISR (known as the Interrupt Vector) is stored in a particular memory location. This location is known as the Vector Address. In the PicoBlaze, the Interrupt Vector is located at `0x3FF`.
- Unlike other processors, note that in PicoBlaze only PC and the Z, C flags are saved during an interrupt process.

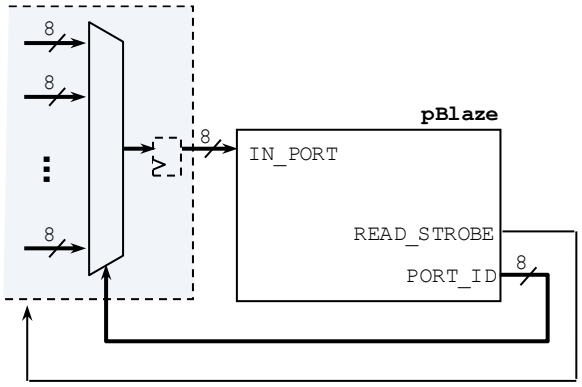


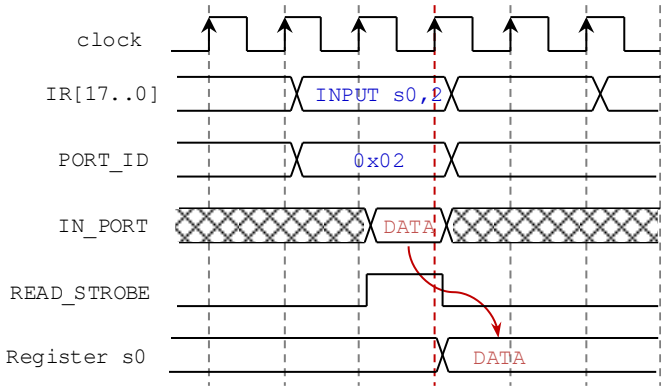
- **I/O Interface:** pBlaze can read/write data from/to external ports via this interface that consists of 5 signals.
  - ✓ I/O Interface signals:
    - IN_PORT, OUT_PORT: 8-bit I/O ports connected to an external interface.
    - PORT_ID: 8-bit port identifier (or port address) for an associated I/O operation. It is valid for 2 clock cycles: this allows enough time for an interface to respond and for reading data from a synchronous RAM.
    - READ_STROBE, WRITE_STROBE: These signals are associated with the read and write operations.

  - ✓ INPUT operation:

`INPUT sX, kk`	PORT_ID ← kk	sX ← IN_PORT
`INPUT sX, (sY)`	PORT_ID ← sY	

    - When the INPUT instruction appears on `IR`, PORT_ID is issued, and it is valid for two clock cycles. After those two cycles, sX captures data on IN_PORT.
    - IN_PORT: It is connected to external interface that allows selection (usually via a multiplexer) of up to 256 different input sources (selected by PORT_ID).
    - READ_STROBE: It is asserted on the second cycle of the two-cycle INPUT instruction cycle. It is used to indicate that pBlaze has acquired data (i.e., acknowledges receipt of data).
    - Example: `INPUT s0,2`.
      Note how PORT_ID appears right after the instruction is issued. Then the external circuit has two cycles to provide an input value (hence we can insert a register before IN_PORT to improve performance). Data is captured on the rising edge right after the second clock cycle of the instruction.





  - ✓ OUTPUT operation:

`OUTPUT sX, kk`	PORT_ID ← kk	OUT_PORT ← sX
`OUTPUT sX, (sY)`	PORT_ID ← sY	

    - When the OUTPUT instruction appears on `IR`, PORT_ID and OUT_PORT are issued and they are valid for two clock cycles. After those two cycles, that data is captured by the external interface.
    - OUT_PORT: It is connected to external interface where a decoder is commonly used (with PORT_ID) to route data onto a specific destination (up to 256 storage spaces, e.g.: registers).
    - WRITE_STROBE: It is asserted on the second cycle of the two-cycle OUTPUT instruction cycle. It indicates data is valid and ready for capture. The external interface can use it as an enable to capture data.
    - Example: `OUTPUT s1,(s9)`
      Note that PORT_ID and OUT_PORT appear immediately after the instruction. Then the external circuit has two cycles to capture the output (hence we can include a pipelining stage in the decoder to improve performance). Data is captured on the rising edge right after the second clock cycle of the instruction.