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

GENERAL 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.
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 in sequence, 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.
```

IR:

```
+-------+-------+-------+-------+-------+
|   OPCODE  |   DR   |   SR   | IMMEDIATE DATA |
+-------+-------+-------+-------+
|      |      |      |        |
|      |      |      |        |
|      |      |      |        |
```

**Figure:** Control Unit and Datapath Unit with single-cycle hardwired control.
### 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: \( M1, M2, M3, M4, M5, M6, L_R0, L_R1, \) and \( L_OP \). Note that the Function Select (FS) output to the ALU is directly generated by \( IR[7..5] \).

- Also, if \( \text{stop ID} = 0 \), all the outputs are '0'. 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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Operation Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>MOV DR, SR</td>
<td>DR ( \leftarrow ) SR</td>
</tr>
<tr>
<td>001</td>
<td>LOADI DR, DATA</td>
<td>DR ( \leftarrow ) DATA, DATA = ( IR[3..0] )</td>
</tr>
<tr>
<td>010</td>
<td>ADD DR, SR</td>
<td>DR ( \leftarrow ) DR + SR</td>
</tr>
<tr>
<td>011</td>
<td>ADDI DR, DATA</td>
<td>DR ( \leftarrow ) DR + DATA, DATA = ( IR[3..0] )</td>
</tr>
<tr>
<td>100</td>
<td>SR0 DR, SR</td>
<td>DR ( \leftarrow ) 0&amp;SR[3..1]</td>
</tr>
<tr>
<td>101</td>
<td>IN DR</td>
<td>DR ( \leftarrow ) IN</td>
</tr>
<tr>
<td>110</td>
<td>OUT DR</td>
<td>OUT ( \leftarrow ) DR</td>
</tr>
<tr>
<td>111</td>
<td>JNZ DR, ADDRESS</td>
<td>PC ( \leftarrow ) PC + 1 if DR=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC ( \leftarrow ) ( IR[3..0] ) if DR( \neq ) 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDRESS = ( IR[3..0] )</td>
</tr>
</tbody>
</table>

#### IN DR:
Dr grabs the contents from the input

```
0 1 1 0 1 1 1 0 1 1 X X X X
IN R0: 10110000 => M1 \( \leftarrow \) 1, L_R0 \( \leftarrow \) 1, M6 \( \leftarrow \) 0
IN R1: 10110000 => M1 \( \leftarrow \) 1, L_R1 \( \leftarrow \) 1, M6 \( \leftarrow \) 0
```

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

```
0 1 1 0 1 1 X X X X
OUT R0: 11100000 => M2 \( \leftarrow \) 0, L_OP \( \leftarrow \) 1, M6 \( \leftarrow \) 0
OUT R1: 11100000 => M2 \( \leftarrow \) 1, L_OP \( \leftarrow \) 1, M6 \( \leftarrow \) 0
```

#### LOADI DR, DATA:
Copies immediate DATA onto DR

```
0 1 0 1 0 1 1 1 1 X X X X d_3 d_2 d_1 d_0
LOADI R0, DATA: 0010d_3d_2d_1d_0 => M5 \( \leftarrow \) 1, M3 \( \leftarrow \) 1, M1 \( \leftarrow \) 0, L_R0 \( \leftarrow \) 1, M6 \( \leftarrow \) 0
LOADI R1, DATA: 0011d_3d_2d_1d_0 => M5 \( \leftarrow \) 1, M3 \( \leftarrow \) 1, M1 \( \leftarrow \) 0, L_R1 \( \leftarrow \) 1, M6 \( \leftarrow \) 0
```

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

```
0 1 0 1 0 1 0 0 X X X X
ADD R0, R0: 01000000 => M4 = 0, M5 = 0, M2 = 0, M3 = 1, M1 = 0, L_R0 = 1, M6 = 0
ADD R0, R1: 01001000 => M4 = 0, M5 = 0, M2 = 1, M3 = 1, M1 = 0, L_R0 = 1, M6 = 0
ADD R1, R0: 01010000 => M4 = 0, M5 = 0, M2 = 1, M3 = 1, M1 = 0, L_R1 = 1, M6 = 0
ADD R1, R1: 01011000 => M4 = 0, 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

```
0 1
0 1 1 DR d3 d2 d1 d0
```

ADDI R0, DATA: 0110d; d3:d2:d1:d0 \(\rightarrow\) M2 \(\leftarrow\) 0, M5 \(\leftarrow\) 1, M3 \(\leftarrow\) 1, M1 \(\leftarrow\) 0, L_R0 \(\leftarrow\) 1, M6 \(\leftarrow\) 0
ADDI R1, DATA: 0111d; d3:d2:d1:d0 \(\rightarrow\) M2 \(\leftarrow\) 1, M5 \(\leftarrow\) 1, M3 \(\leftarrow\) 1, M1 \(\leftarrow\) 0, L_R1 \(\leftarrow\) 1, M6 \(\leftarrow\) 0

**MOV DR, SR:** Copies the contents of SR onto DR

```
0 1 0 1
0 0 0 0 DR SR X X X
```

MOV R0, R0: 00000XXX \(\rightarrow\) M2 \(\leftarrow\) 0, M3 \(\leftarrow\) 0, M1 \(\leftarrow\) 0, L_R0 \(\leftarrow\) 1, M6 \(\leftarrow\) 0
MOV R1, R1: 00011XXX \(\rightarrow\) M2 \(\leftarrow\) 1, M3 \(\leftarrow\) 0, M1 \(\leftarrow\) 0, L_R1 \(\leftarrow\) 1, M6 \(\leftarrow\) 0
MOV R0, R1: 00001XXX \(\rightarrow\) M2 \(\leftarrow\) 1, M3 \(\leftarrow\) 0, M1 \(\leftarrow\) 0, L_R0 \(\leftarrow\) 1, M6 \(\leftarrow\) 0
MOV R1, R0: 00010XXX \(\rightarrow\) M2 \(\leftarrow\) 0, M3 \(\leftarrow\) 0, M1 \(\leftarrow\) 0, L_R1 \(\leftarrow\) 1, M6 \(\leftarrow\) 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

```
0 1 0 1
1 0 0 0 DR SR X X X
```

SR0 R0,R0: 10000XXX \(\rightarrow\) M4 \(\leftarrow\) 0, M5 \(\leftarrow\) 0, M2 \(\leftarrow\) 0, M3 \(\leftarrow\) 1, M1 \(\leftarrow\) 0, L_R0 \(\leftarrow\) 1, M6 \(\leftarrow\) 0
SR0 R0,R1: 10001XXX \(\rightarrow\) M4 \(\leftarrow\) 0, M5 \(\leftarrow\) 0, M2 \(\leftarrow\) 1, M3 \(\leftarrow\) 1, M1 \(\leftarrow\) 0, L_R0 \(\leftarrow\) 1, M6 \(\leftarrow\) 0
SR0 R1,R0: 10010XXX \(\rightarrow\) M4 \(\leftarrow\) 0, M5 \(\leftarrow\) 0, M2 \(\leftarrow\) 1, M3 \(\leftarrow\) 1, M1 \(\leftarrow\) 0, L_R1 \(\leftarrow\) 1, M6 \(\leftarrow\) 0
SR0 R1,R1: 10011XXX \(\rightarrow\) M4 \(\leftarrow\) 0, M5 \(\leftarrow\) 0, M2 \(\leftarrow\) 1, M3 \(\leftarrow\) 1, M1 \(\leftarrow\) 0, L_R1 \(\leftarrow\) 1, M6 \(\leftarrow\) 0

**JNZ DR, ADDRESS:** Jumps to a certain instruction if DR \(\neq 0\). This is how computers implement loops.

```
0 1
1 1 1 DR a3 a2 a1 a0
```

JNZ R0, ADDRESS: 1110a3:a2:a1:a0 \(\rightarrow\) M2 \(\leftarrow\) 0, M6 \(\leftarrow\) 0 if \(z = 1\), M6 \(\leftarrow\) 1 if \(z = 0\)
JNZ R1, ADDRESS: 1111a3:a2:a1:a0 \(\rightarrow\) M2 \(\leftarrow\) 1, M6 \(\leftarrow\) 0 if \(z = 1\), M6 \(\leftarrow\) 1 if \(z = 0\)

* \(M6 \leftarrow 0 = PC \leftarrow PC + 1\); \(M6 \leftarrow 1 = PC \leftarrow IR[3..0]\)

**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 if \(FS=101,110\) (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 \(\leftarrow\) 1
out R0 OUT = 1
addi R0,1 R0 \(\leftarrow\) R0 + 1 = 2
out R0 OUT = 2
addi R0,1 R0 \(\leftarrow\) R0 + 1 = 3
out R0 OUT = 3
addi R0,1 R0 \(\leftarrow\) R0 + 1 = 4
out R0 OUT = 4
addi R0,1 R0 \(\leftarrow\) 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.

```
* 2 to 13 = 4 to 15

start: load R0,2    R0 ← 2
       load 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.
- 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: \( N_I = 16, N = 16, K = H = 6, M = 3 \) (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 (\( K \leq H \)).
- 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.: \( 111110 + 000011 = 10000001 \); \( 000001 - 000011 = 000000 + 111101 = 111110 \)).
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.

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

- 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 \(2^{16}=64\) 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 \(2^{16}=64\) 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 computer 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 Unit 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 microoperations, 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 of 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Description</th>
<th>PC</th>
<th>Status Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement</td>
<td>0000110</td>
<td>DEC</td>
<td>RD, RA</td>
<td>R[DR] ← R[SA]- 1</td>
<td>PC ← PC+1</td>
<td>N, Z, C, V</td>
</tr>
<tr>
<td>NOT</td>
<td>0001011</td>
<td>NOT</td>
<td>RD, RA</td>
<td>R[DR] ← not (R[SA])</td>
<td>PC ← PC+1</td>
<td>N, Z</td>
</tr>
<tr>
<td>Move B</td>
<td>0001100</td>
<td>MOVB</td>
<td>RD, RA</td>
<td>R[DR] ← R[SB]</td>
<td>PC ← PC+1</td>
<td>N, Z</td>
</tr>
<tr>
<td>Shift Right</td>
<td>0001110</td>
<td>SHR</td>
<td>RD, RB</td>
<td>R[DR] ← sl R[SB]</td>
<td>PC ← PC+1</td>
<td>N, Z</td>
</tr>
<tr>
<td>Shift Left</td>
<td>0001111</td>
<td>SHL</td>
<td>RD, RB</td>
<td>R[DR] ← sl R[SB]</td>
<td>PC ← PC+1</td>
<td>N, Z</td>
</tr>
<tr>
<td>Load Immediate</td>
<td>0101100</td>
<td>LDI</td>
<td>RD, OP</td>
<td>R[DR] ← OP</td>
<td>PC ← PC+1</td>
<td>N, Z</td>
</tr>
<tr>
<td>Store</td>
<td>0100000</td>
<td>ST</td>
<td>RA, RB</td>
<td>M[R[SA]] ← R[SB]</td>
<td>PC ← PC+1</td>
<td></td>
</tr>
<tr>
<td>Branch on Zero</td>
<td>1100000</td>
<td>BRZ</td>
<td>RA, AD</td>
<td>If R[SA] ≠ 0</td>
<td>PC ← PC+1</td>
<td>N, Z</td>
</tr>
<tr>
<td>Brand on Negative</td>
<td>1100001</td>
<td>BRN</td>
<td>RA, AD</td>
<td>If R[SA] &lt; 0</td>
<td>PC ← PC+1</td>
<td>N, Z</td>
</tr>
<tr>
<td>Jump</td>
<td>1110000</td>
<td>JMP</td>
<td>RA</td>
<td>PC ← R[SA]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other ISAs do not generate status bits when transfers on the Bus 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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory Contents</th>
<th>Other Fields</th>
<th>Assembly Instruction</th>
<th>Operation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>100 100 01 1000000</td>
<td></td>
<td>start: LDI R3,4</td>
<td>R3 ← 4</td>
<td></td>
</tr>
<tr>
<td>0000001</td>
<td>100 100 00 010001</td>
<td></td>
<td>LDI R0,2</td>
<td>R0 ← 2</td>
<td></td>
</tr>
<tr>
<td>0000010</td>
<td>100 100 00 1110000</td>
<td></td>
<td>LDI R1,7</td>
<td>R1 ← 7</td>
<td></td>
</tr>
<tr>
<td>0000011</td>
<td>100 100 01 0110000</td>
<td></td>
<td>ADI R1,R1,4</td>
<td>R1 ← R1+4</td>
<td></td>
</tr>
<tr>
<td>0000100</td>
<td>100 100 00 0000000</td>
<td></td>
<td>loop: ADI R0,R0,1</td>
<td>R0 ← R0+1</td>
<td></td>
</tr>
<tr>
<td>0000101</td>
<td>000 011 01 0000000</td>
<td></td>
<td>DEC R1,R1</td>
<td>R1 ← R1-1</td>
<td></td>
</tr>
<tr>
<td>0000110</td>
<td>111 000 11 1110000</td>
<td></td>
<td>BRZ R1,-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000111</td>
<td>111 000 00 0110000</td>
<td></td>
<td>JMP R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000100</td>
<td>000 000 00 0000000</td>
<td></td>
<td>R0 ← R0 (This is NOP operation)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Memory</th>
<th>Assembly Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>100 100 01 1000000</td>
<td>start: LDI R3,4</td>
</tr>
<tr>
<td>0000001</td>
<td>100 100 00 010001</td>
<td>LDI R0,2</td>
</tr>
<tr>
<td>0000010</td>
<td>100 100 00 1110000</td>
<td>LDI R1,7</td>
</tr>
<tr>
<td>0000011</td>
<td>100 100 01 0110000</td>
<td>ADI R1,R1,4</td>
</tr>
<tr>
<td>0000100</td>
<td>100 100 00 0000000</td>
<td>loop: ADI R0,R0,1</td>
</tr>
<tr>
<td>0000101</td>
<td>000 011 01 0000000</td>
<td>DEC R1,R1</td>
</tr>
<tr>
<td>0000110</td>
<td>111 000 11 1110000</td>
<td>BRZ R1,-5</td>
</tr>
<tr>
<td>0000111</td>
<td>111 000 00 0110000</td>
<td>JMP R3</td>
</tr>
<tr>
<td>0000100</td>
<td>000 000 00 0000000</td>
<td>R0 ← R0 (This is NOP operation)</td>
</tr>
</tbody>
</table>
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=0, all the outputs are '0'. 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:

<table>
<thead>
<tr>
<th>Instruction Register</th>
<th>V</th>
<th>C</th>
<th>N</th>
<th>Z</th>
<th>RW</th>
<th>DR</th>
<th>SA</th>
<th>SB</th>
<th>MB</th>
<th>MD</th>
<th>FS</th>
<th>MW</th>
<th>OS</th>
<th>JS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVA R3, R2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>11</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>MOVA R7, R0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>MOVB R0, R3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>MOVB R6, R6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>ADD R6, R0, R0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>XOR R6, R1, R3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>XOR R5, R4, R5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>LDI R7, 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>LDI R5, 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>ADI R6, R6, 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>LD R3, R7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>ST R1, R5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>BRN R4, -5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>BRZ R3, 12</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>JMP R5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

- 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.
- 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, DDRAMs, 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.