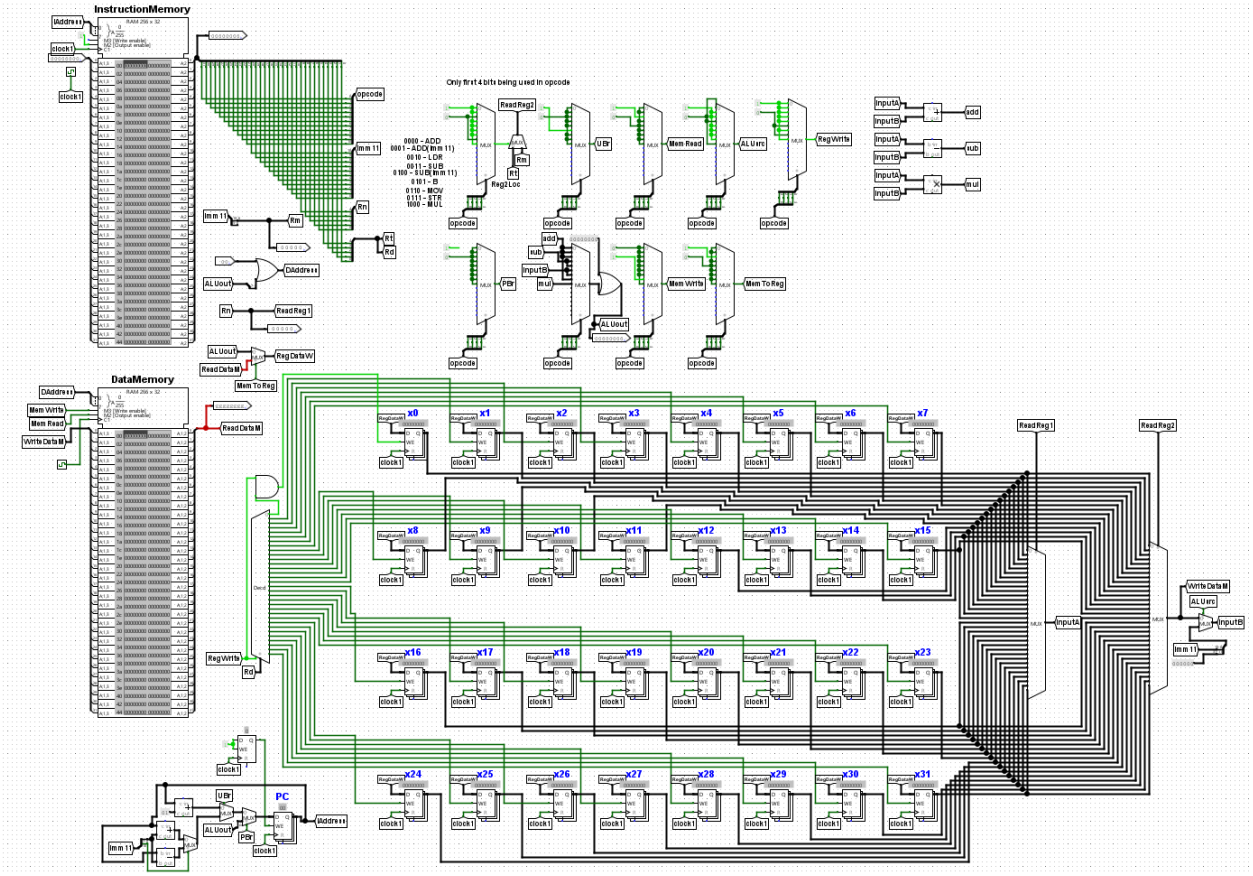
CPU Name: The Menace

Author: Jacob Rosengarten

0. Broad Visual



1. Usage:

Instructions can be either loaded or edited into the instruction memory in the top left. Data can be loaded, edited, or inserted via instructions in the bottom left of the CPU. The instructions and data are stored as hexadecimals but the importance is the sequences of 0s and 1s in binary.

Creating an Instruction:

Instructions are made up of 32-bits. See the tables below for the 2 types of instructions.

31 ---------- 21	20 ---------- 16	15 ---------- 10	9 ---------- 5	4 ---------- 0
opcode	Rm	111000	Rn	Rd

Table 1.1: Type-1 instruction format

31 ---------- 21	20 ---------- 10	9 ---------- 5	4 ---------- 0
opcode	11-bit Immediate	Rn	Rd/Rt

Table 1.1: Type-2 instruction format

Instructions are formulated using opcode to determine the operation as well as register numbers to determine the operands and destination. The first type of instruction (Table 1.1) contains two source registers and a destination register while the second type of instruction (Table 1.2) uses an immediate value instead of a second register.

```
ADD X0, X1, X2
SUB X0, X1, X2
MUL X0, X1, X2
```

Figure 1.3: Type-1 instruction examples.

```
ADD X0, X1, 30
SUB X0, X1, 25
```

Figure 1.4: Type-2 instruction examples.

The address of the first source register is written to Rn. This value is present in all instructions except branching instructions. There are 32 different registers to use for immediate data access from X0 to X31. The five bits 9 through 5 represent Rn. Bits 20 through 10 determine the second input via register 2, which has its address stored into bits 20 through 16 as Rm, or an immediate 11-bit value. The last 5 bits (4 through 0) are used for the destination (Rd) or target register (Rt). Each register address uses five bits to represent the 0 through 31 using unique bit combinations. Branching/memory accessing instructions use Rt and not Rd.

For the current version of this CPU, only the first 4 bits will affect the opcode meaning that bits 27 through 21 have no functionality and are currently fillers as there are no more than 16 instructions.

Below is a table of the opcode values:

Type-1 Instructions	
ADD	00000000000
SUB	00110000000
MUL	10000000000

Table 1.5: Type-1 instruction opcodes

Type-2 Instructions	
ADD	00010000000
SUB	01000000000
LDR	00100000000
B	01010000000
MOV	01100000000
STR	0110000000

Table 1.6: Type-2 instruction opcodes

The branching instruction B works differently from the other Type-2 instructions. When using B, only the immediate 11-bit value is needed for functionality. This value is the relative offset between the PC and where the PC should reach via this instruction.

These instructions are based on ARM-64 but use different opcodes.

Upon creating an instruction, it can be loaded into the machine as an 32-bit 8 character hexadecimal. Instructions should be loaded in order from address 0 incrementing by 1.

The state of the instruction memory can be saved and loaded from a file.

A template instruction file is attached named dotprod

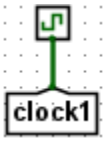
Data Loading:

Loading data is done by instruction or manual insertion into the Random Access Memory (RAM). Like the instructions, the data is stored in 32 bits. This can be useful to store addresses and values. The STR instruction stores data from the registers onto the data RAM and the LDR instruction reads data from the RAM and stores it onto a register. Data values are stored and read as a hexadecimal.

A template data file is attached named as dotprodD

2. Architecture

Overview:



This CPU uses two clock cycles. The main clock cycle (clock1) phases last for two ticks while the data clock cycle phases change every tick. All data used and stored in this PC is 32-bit with the exception of the instruction address and the data address used to access instructions and data from the RAM.

Instruction Fetching and Decoding:

The program counter(PC) contains the address of the current instruction to be read by the instruction memory.

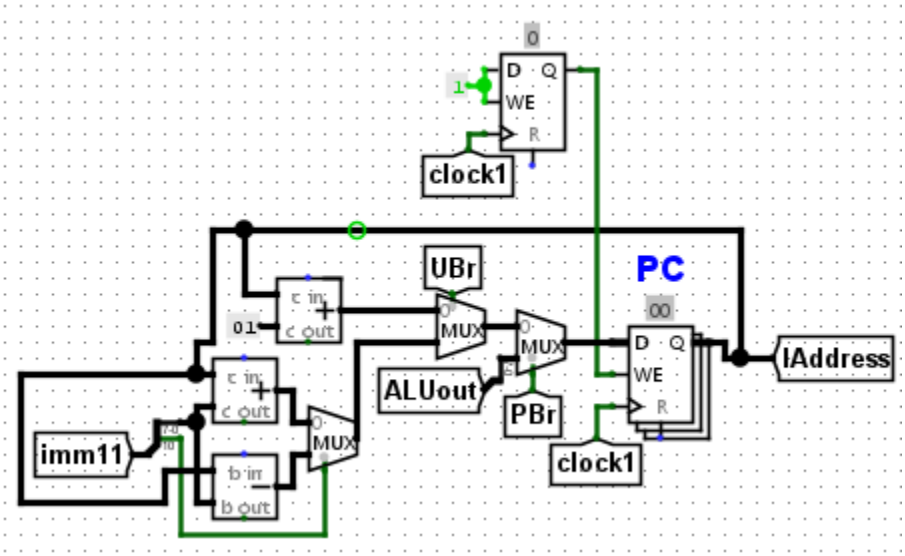


Figure 1.7: Instruction Fetching unit.

To determine the next instruction, a few multiplexors are used against a few parameters. UBr is only enabled when using the B instruction and PBr is not implemented. Therefore, PC increments each clock cycle where B is not currently executing. When B is in use, the immediate value is added to the current instruction address. The first bit of the immediate value is used to determine whether the value is positive or negative so a multiplexor is used to determine whether addition or subtraction is used. A secondary register above is used for the write-enable port to buffer adding to the address in order to ensure the initial instruction is read.

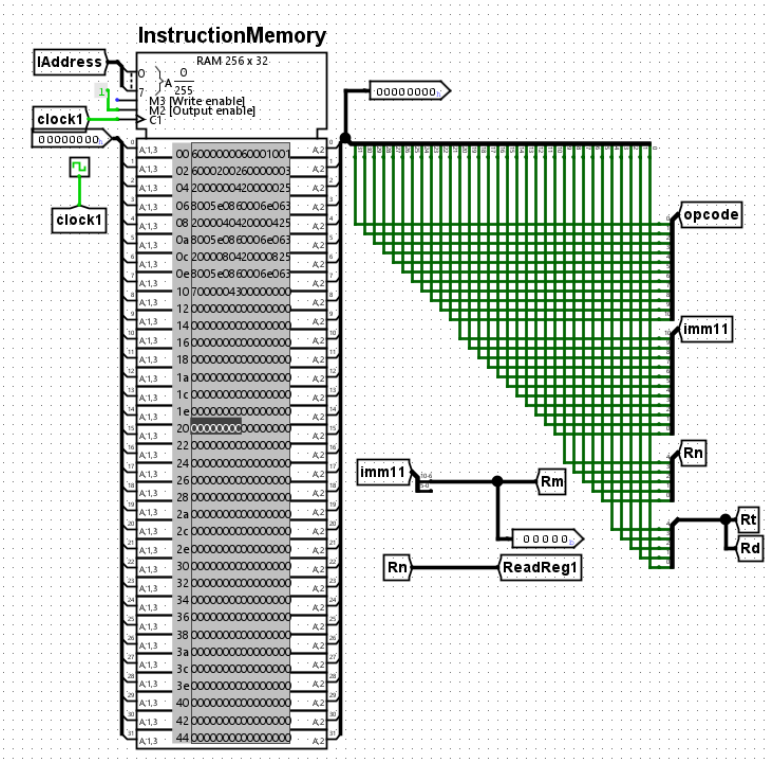


Figure 1.8: Instruction Memory and Decoding unit

The instruction memory holds all of the instructions which are then read from the RAM and dissected into multiple variables which will then be sent to the Arithmetic Logic Unit and the registers' input controls.

Arithmetic Logic Unit:

This CPU uses multiplexors using the opcode as the select bit in order to control different variable wires.

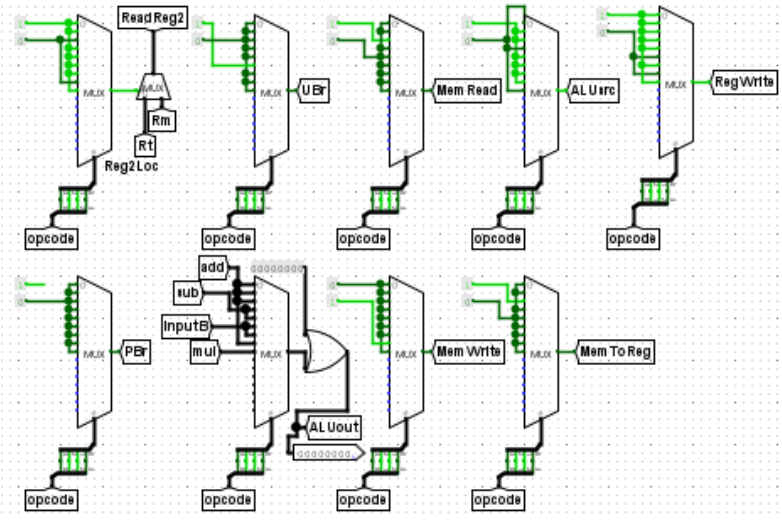


Figure 1.9: Multiplexor Control Unit

As shown by Figure 1.9, only the first four bits are used in the opcode to determine which of the nine implemented instructions to use as a selector. In this current implementation, up to 16 instructions can be loaded at once but more bits can be used from the opcode in the case that more than 16 instructions are to be implemented.

One of the most important multiplexors used is the one that determines ALUout. Without this multiplexor, the data resulting from all operations would be voided. In most instructions, ALUout links directly to RegDataW which is then written into the registers. In other cases, ALUout determines the addresses for the Data Memory unit.

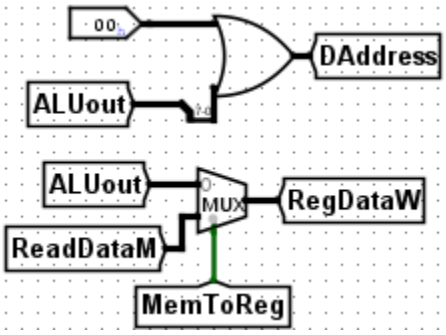


Figure 1.10: The last 8 bits of ALUout are used to address the Data Memory unit. When the memory unit is not being read from to a register using LDR, ALUout leads to RegDataW which leads to the registers.

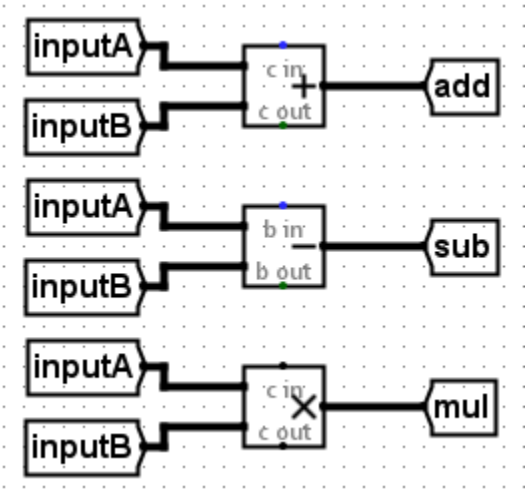


Figure 1.11: The operations are executed using an adder, subtractor, and multiplier which are then sent to the ALUout multiplexor to be then written to a register.

Data Memory and Registers:

The registers in the CPU hold data that is readily available for operations. On the contrary, the Data Memory unit stores data that can be read and written to but cannot have operations directly operated upon without the use of registers.

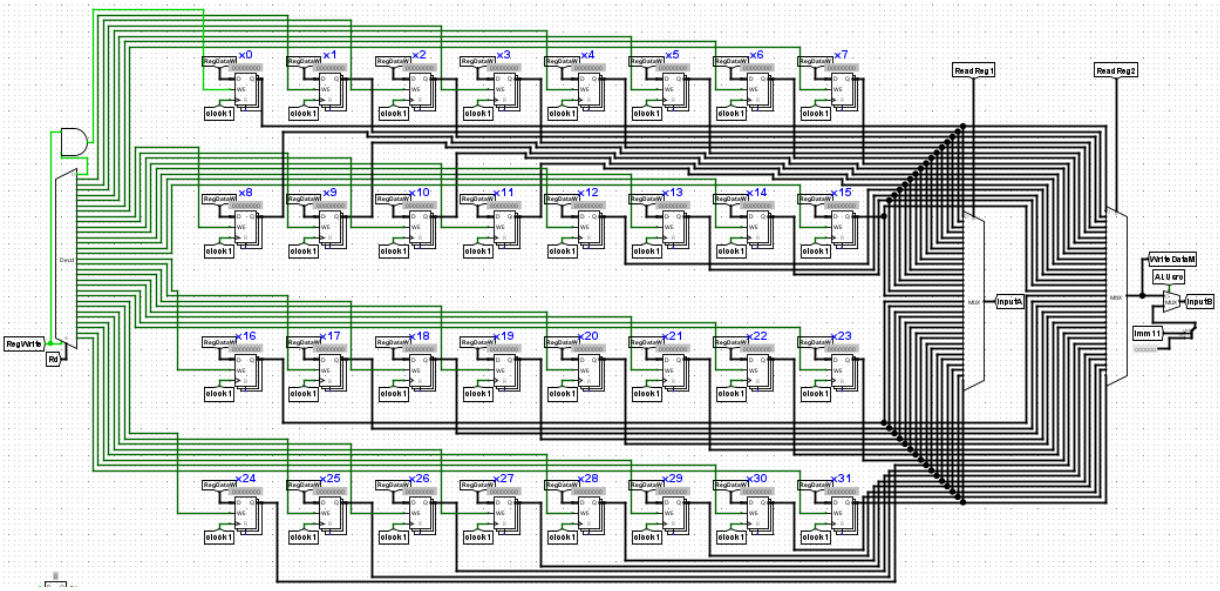


Figure 1.12: The 32 32-bit registers are necessary for any usage of the CPU as every non-branching instruction requires at least one register for reasonable use.

When the registers are written to, RegDataW holds the data to be written into the register which funnels into all 32 registers. The write-enable ports (WE) determine whether or not the register is able to be written to in the given moment. RegWrite, from the ALU and control unit, enables a decoder to toggle one register's WE port which is decided by Rt.

When the registers are read from, ReadReg1 and ReadReg2 determine which register's values are stored into inputA and inputB respectively. When an immediate value is used instead of a second register, ALUsrc will be toggled and the immediate value is put into inputB instead of the value from a second register. When data is written to the memory, the memory is read from the second register then written from WriteDataM to the Data Memory Unit.

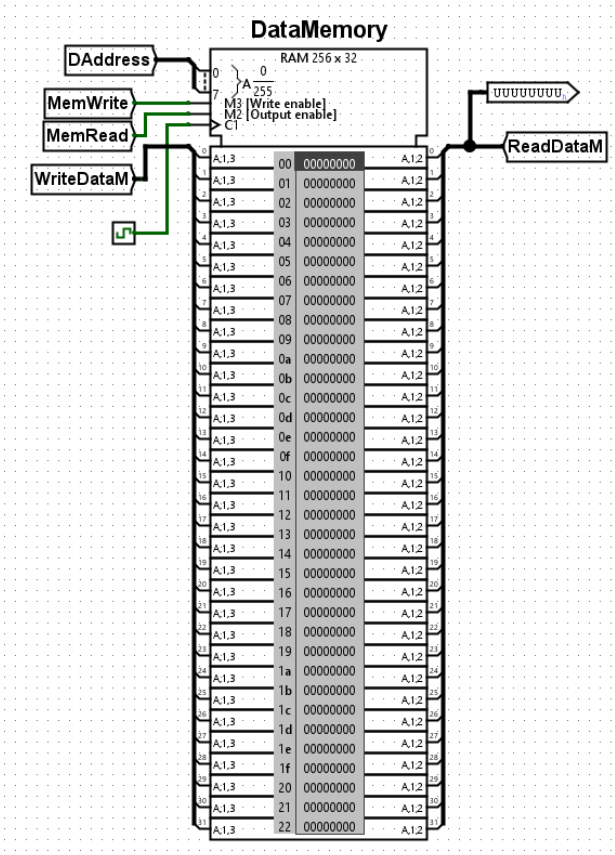


Figure 1.9: Data Memory Unit

3. Example: Dot Product

Overview:

Two example files are included with this manual.

dotprod - File containing instructions for the dotprod program

dotprodD - File containing data for the dotprod program

To Run:

1. Open the .circ file containing the CPU or Reset Simulation(Ctrl + R)
2. Right click on the instruction memory then select “Load Image...”
    a. Navigate to the containing folder and select “dotprod”
3. Right click on the data memory then select “Load Image...”
    a. Navigate to the containing folder and select “dotprod.d”
4. Run the CPU by either manually ticking(Ctrl + T) or enable auto-tick(Ctrl + K)

Executed instructions:

```
MOV X0 X0 0 # X0 = vec1 address
MOV X1 X0 4 # X1 = vec2 address
MOV X2 X0 8 # X2 = dot add
MOV X3 X0 0 # X3 = 0 (res)

LDR X4 [X0 0] # X4 = vec1[0]
LDR X5 [X1 0] # X4 = vec2[0]
MUL X6 X4 X5 # X6 = vec1[0]*vec2[0]
ADD X3 X3 X6 # res += vec1[0]*vec2[0]

LDR X4 [X0 1] # X4 = vec1[1]
LDR X5 [X1 1] # X4 = vec2[1]
MUL X6 X4 X5 # X6 = vec1[1]*vec2[1]
ADD X3 X3 X6 # res += vec1[1]*vec2[1]

LDR X4 [X0 2] # X4 = vec1[2]
LDR X5 [X1 2] # X4 = vec2[2]
MUL X6 X4 X5 # X6 = vec1[2]*vec2[2]
ADD X3 X3 X6 # res += vec1[2]*vec2[2]

STR X3 [X2 0] # Store res into dot
```

Results:

By default, vec1 = 10, 20, 30 and vec2 = 11, 22, 33. This data can be altered in dotprodD or manually into the data memory before execution. The result is stored into address 8.

After executing dotprod using the default values, the hexadecimal result in address 8 is 604.

10*11 + 20*22 + 30*33 = 1540

6*16^2 + 0*16 + 4 = 1540

Thank you for reading this far!