IKEA

It’s called IKEA because you need to assemble everything yourself.

Created by: Justin Chen

Specifications

All instructions are 8 bytes and are stored in ROM.
All data stored in memory are byte-addressed and stored in RAM.

There are 32 general purpose registers, ranging from X0 to X31. These registers are 8 bits.
It is, however, highly discouraged to write to X30 manually as that stored the program counter memory address when branching.
X31 will always have a value of 0x0. This value cannot be changed.

This is not a pipelined program. However, it can handle branching-related instructions and generates separate image files for .data and .text directives.

IKEA Instructions

Instruction Table Notation

CONTROL - Control Codes

- These encapsulate the control flags

ALUCODE - Arithmetic Codes

- These encapsulates the ALU control flags

OPCODE - Operation Code

- This tells the CPU what operation we're taking

WR - Write Register

- The register to write to (if any)

RR1 - Read Register 1

- The 1st register to read

RR2 - Read Register 2

- The 2nd register to read from

8 Bit Immediate - 8 Bit unsigned integer to pass into the command

- Can handle up to 255. Any number after 255 will be truncated

Label - IKEA Assembly Code Labels

- Syntax: `labelName:`
- These are pre-processed by the IKEA Assembler

Control Code Embeddings

CONTROL[0]	Determines whether the CPU sets the zero flag

CONTROL[1]	Determines whether the CPU uses immediate numbers instead of reading registers.
CONTROL[2]	Determines whether the CPU reads memory
CONTROL[3]	Determines whether the CPU writes to memory from output (STORE)
CONTROL[4]	Determines whether the CPU writes to registers from ALU
CONTROL[5]	Determines whether the CPU writes to registers from memory reading (LOAD)
CONTROL[6]	Determines whether the CPU does conditional branching
CONTROL[7]	Determines whether the CPU does unconditional branching
CONTROL[8]	Determines whether the CPU puts X30 into PC
CONTROL[9]	Determines whether the CPU reads WR as RR2.
CONTROL[10]	Determines whether the CPU puts PC + 8 into X30

ALU Code Embeddings

These ALUCODEs are mutually exclusive. When one ALUCODE is on, the others should be off (already built into the architecture).

ALUCODE[0]	Determines whether the ALU Adds
ALUCODE[1]	Determine whether the ALU Subtracts
ALUCODE[2]	Determines whether the ALU performs an AND

ALUCODE[3]	Determines whether the ALU performs an OR
ALUCODE[4]	Do nothing (passes zero)
ALUCODE[5]	Pass the value in register 2 or immediate number (whichever comes in)
ALUCODE[6]	Pass the value in register 1
ALUCODE[7]	Pass the negated value in register 1

All instruction binary encodings are 64 bits, since:

- Control codes are 16 bits
- ALU codes are 8 bits
- OPCODEs are 16 bits
- RR2, RR1, and WR are all 8 bits

Arithmetic Instructions

We will use X to denote don't care. These would, in practice, be filled with 0.

Registers numbers RR1, RR2, and WR cap at 5 bytes. They go from X0 - X31. However:

- X30 stores PCs when executing BRANCH_LINK. It is generally *not* advised to update this manually
- X31 is immutable and will always have a value of 0

	63 ↔ 48	47 ↔ 40	39 ↔ 24	23 ↔16	15 ↔ 8	7 ↔ 0
Instruction	CONTROL	ALUCODE	OPCODE	RR2	RR1	WR

ADD WR, RR1, RR2 Adds values in RR1 and RR2 and stores them into WR	0000100000000000	10000000	1000000000000000			
SUB WR, RR1, RR2 Subtracts values in RR1 and RR2, and sets it to WR (RR1 - RR2)	0000100000000000	01000000	0010000000000000			
AND WR, RR1, RR2 Does bitwise and on RR1 and RR2, and sets it to WR	0000100000000000	00100000	0000100000000000			
OR WR, RR1, RR2 Does bitwise or on RR1 and RR2, and sets it to WR	0000100000000000	00010000	0000010000000000			

Memory Instructions

	63 ↔ 48	47 ↔ 40	39 ↔ 24	23 ↔16	15 ↔ 8	7 ↔ 0
Instruction	CONTROL	ALUCODE	OPCODE	8 Bit Immediate #	RR1	WR
LOAD WR, RR1, 8 bit immediate (offset)	0110010000000000	10000000	0000001000000000			

MAX_INT = 255 Loads the item in the address stored in RR1 + 8 bit immediate offset and stores it in WR						
ADDRESS WR, label Puts the address of the data (represented by the label) into WR The label will be converted to an 8 bit immediate # that represents the memory address of label	0100100000000000	00000100	00000000000000 01		X	

	63 ↔ 48	47 ↔ 40	39 ↔ 24	23 ↔16	15 ↔ 8	7 ↔ 0
Instruction	CONTROL	ALUCODE	OPCODE	8 Bit Immediate #	RR1	RR2
STORE RR2, RR1, 8 bit immediate (offset) Stores the value in RR2 into memory with the address stored in RR1 + 8 bit offset We don't write to WR in this	0101000001000000	10000000	00000001000000 00			

command. We will read it. In addition, the ALU performs RR1 + 8 bit immediate like LOAD does MAX_INT = 255						

Register To Register Instructions

Instruction	CONTROL	ALUCODE	OPCODE	RR2	RR1	WR
	63 ↔ 48	47 ↔ 40	39 ↔ 24	23 ↔ 16	15 ↔ 8	7 ↔ 0
SET WR, RR2 Sets WR to the value stored in RR2	0000100000000000	00000100	0000000010000000		X	

Instruction	CONTROL	ALUCODE	OPCODE	8 Bit Immediate #	RR1	WR
	63 ↔ 48	47 ↔ 40	39 ↔ 24	23 ↔ 16	15 ↔ 8	7 ↔ 0
SETIMM WR, 8 bit immediate Sets WR to the 8 bit immediate	0100100000000000	00000100	0000000001000000		X	

^^ Don't care

Branching Instructions

Label will be converted to the instruction address when running these instructions.

This is done by the IKEA Assembler calculating the instruction offsets and putting them in as 8 bit immediate numbers.

- Remember - each instruction is 8 bytes. Take that into account when calculating offsets

Final offset calculations are done in the PCC_Updater and not in ALU. It directly retrieves offset input from the 8 Bit immediate number.

	63 ↔ 48	48 ↔ 40	40 ↔ 25	24 ↔17	16 ↔ 9	8 ↔ 0
Instruction	CONTROL	ALUCODE	OPCODE	8 Bit Immediate # (offset)	RR1	WR
BRANCH label Unconditionally branches to instruction address represented by label	0100000100000000	00001000	0000000000100000		X	X
BRANCH_LINK label Does BRANCH, but sets X30 to current program + 4	0100000100100000	00001000	000000000001000 0		X	00011110 (X30)
BRANCH_IF_ZERO RR1, label Does BRANCH if the value in RR1 is 0	0100001000000000	00000010	000000000000100 0			X
BRANCH_IF_NOT_ZERO RR1, label	0100001000000000	00000001	000000000000010 0			X

Does BRANCH if the value in RR1 is not 0						

Instruction	CONTROL	ALUCODE	OPCODE	RR2	RR1	WR
RETURN (the return address will be forced to be X30) Sets the program counter to the value in X30	0000000010000000	00000100	0000000000000010	00011110 (X30)	X	X

Executing Program

Here's the pipeline/sequence for how an IKEA program is run:

IKEA Assembly File → IKEA Image Files → IKEA CPU

IKEA Assembly File

The IKEA Assembly file holds a sequence of IKEA Instructions.
These are executed from the top down.

Here's a sample program:

```
.text

ADDRESS X0, donut            # Fetch the address of donut and put it in X0
ADDRESS X1, jumbo            # Fetch the address of jumbo and put it in X1

LOAD X2, X0, 0               # Loads in donut from memory
LOAD X3, X1, 0               # Loads in jumbo from memory

SUB X4, X3, X2               # X4 = X3 - X2
BRANCH_IF_ZERO X4, _amogus           # If X4 is zero, jump to _amogus

STORE X4, X0, 0              # Stores X3 - X2 into donut from memory if they're not equal (difference isn't 0)
BRANCH _end                  # Jumps to the end

_amogus:
ADD X5, X3, X2               # X5 = X3 + X2
STORE X5, X0, 0              # Stores X3 + X2 into donut from memory if they're equal

_end:
SETIMM X6, 8                    # Set X6 to 8

.data
donut: 5
jumbo: 5
```

Other than that, here's the syntax you need to know:

- IKEA Assembly files end in .ikea
- Comments are denoted with #
- The directive .text must come before .data
    - .text contains all the IKEA Instructions

- ○ .data contains all the data to be stored in memory. Remember this is byte-addressed and holds up to 1 byte.
  - ○ To declare something in .data, use the syntax `item_name:value`
- To declare a label, use the syntax `label_name:`
- Remember that each IKEA instruction is 8 bytes. Due to how Logisim RAMs are set up, the number of instructions you are able to put into the IKEA Assembly file may be limited

IKEA Image Files

Once you are done writing the IKEA Assembly File, you can cross-assemble it with the IKEA Assembler.
The Assembler file is called ikeaAssemble.py

Here's the syntax (to see this screen, run `$python ikeaAssemble.py -h`):

```
usage: $python ikeaAssemble.py -f [file_path] -a [Write path for .data] -o [Write path for .text]

Assembles a IKEA file and generates binary codes according to specifications. The file must end in .ikea

options:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  The code file to assemble
  -a RAM, --ram RAM     Write path for .data image file
  -o ROM, --rom ROM     Write path for .text image file
```

All 3 parameters are required.

Here's an example of an Assemble command being run. Note that all the paths are relative.

```
python .\ikeaAssemble.py -f .\compare.ikea -a ./RAM.txt -o ./ROM.txt
```

Upon assembling, you will see two image files.

The “RAM.txt” file (you might call it something else) contains everything in the .data directive. This will be loaded into DataMemory.

RAM.txt

```
v3.0 hex words addressed
00: 05 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The “ROM.txt” file (you might call it something else) contains everything in the .text directive. This will be loaded into InstructionMemory. These are in hexadecimal.

ROM.txt

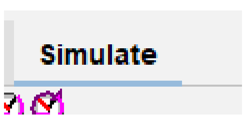
```
v3.0 hex words addressed
00: 00 00 00 01 00 04 00 48 01 00 01 01 00 04 00 48
10: 02 00 00 00 02 80 00 64 03 01 00 00 02 80 00 64
20: 04 03 02 00 20 40 00 08 00 04 18 08 00 02 00 42
30: 04 00 00 00 01 80 40 50 00 00 18 20 00 08 00 41
40: 05 03 02 00 80 80 00 08 05 00 00 00 01 80 40 50
50: 06 00 08 40 00 04 00 48 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

IKEA CPU

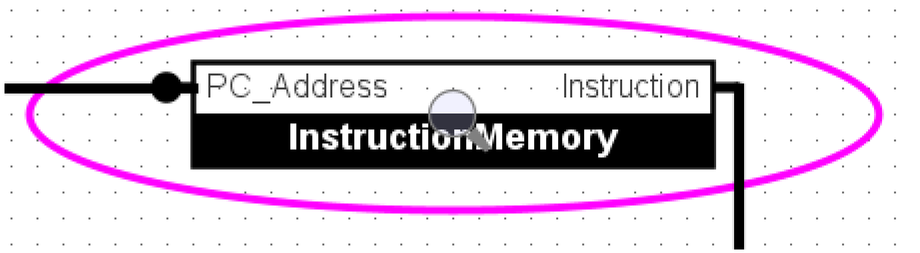
After retrieving the image files, load them onto the CPU.

The file should be called ikea.circ

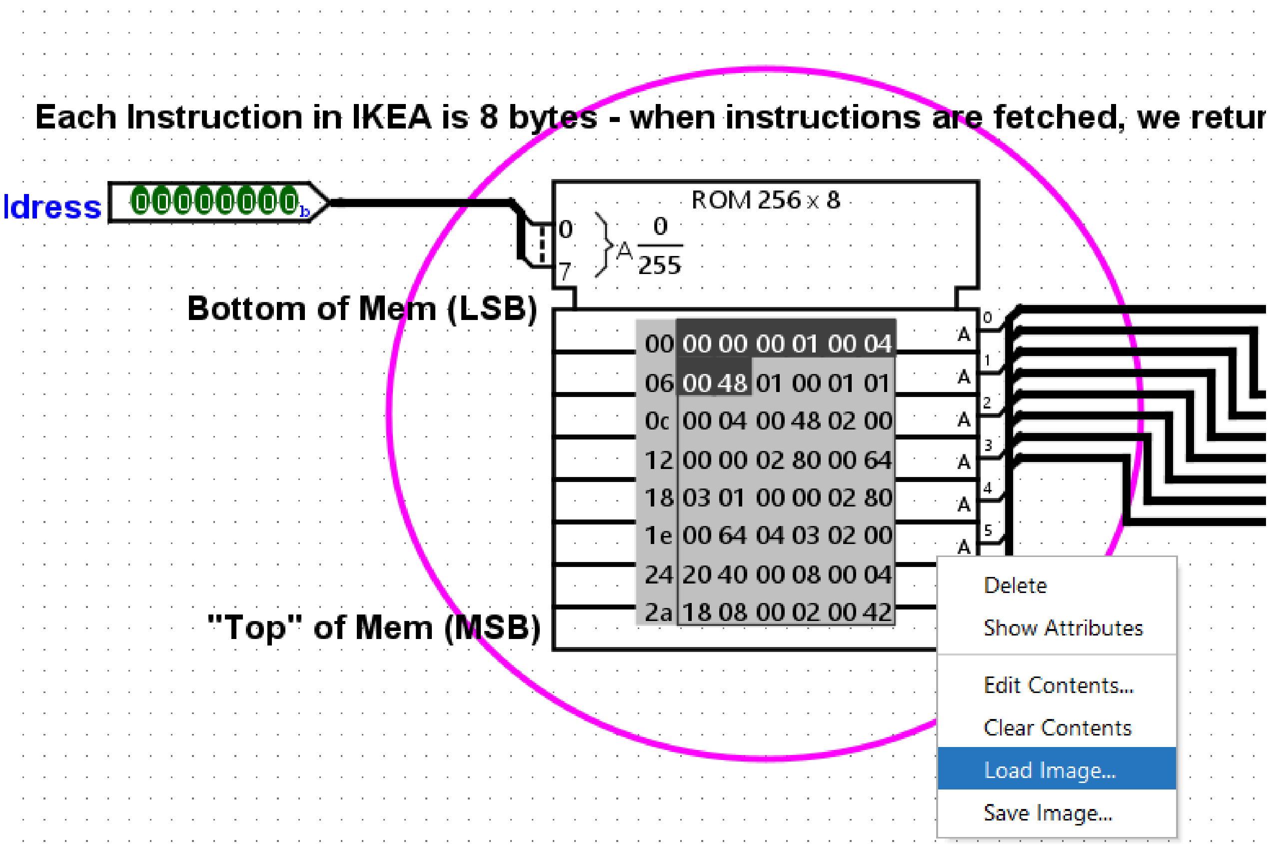
Click simulate on the left panel.



Click on the instruction memory and open the magnifying glass.

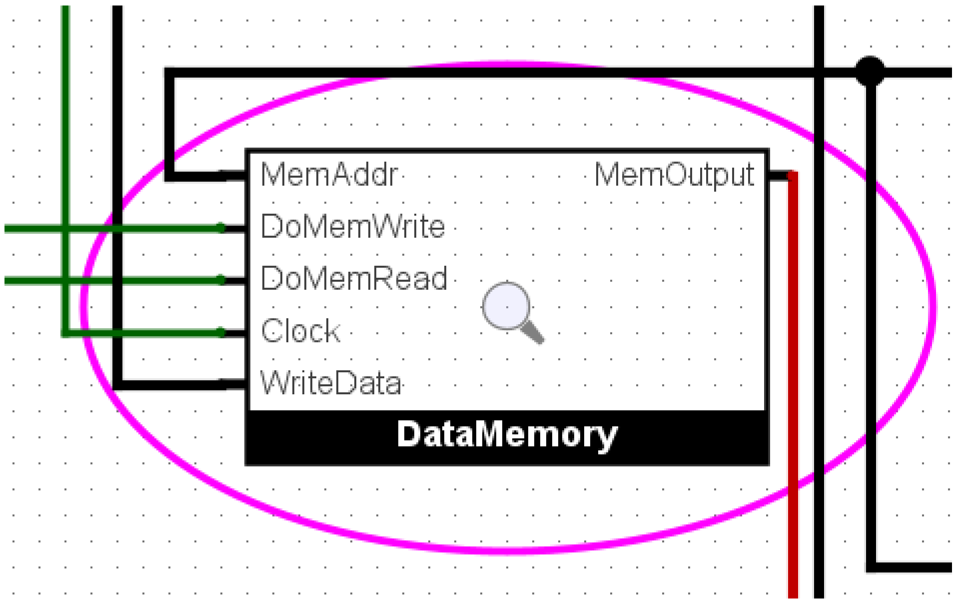


And load the instruction image file.

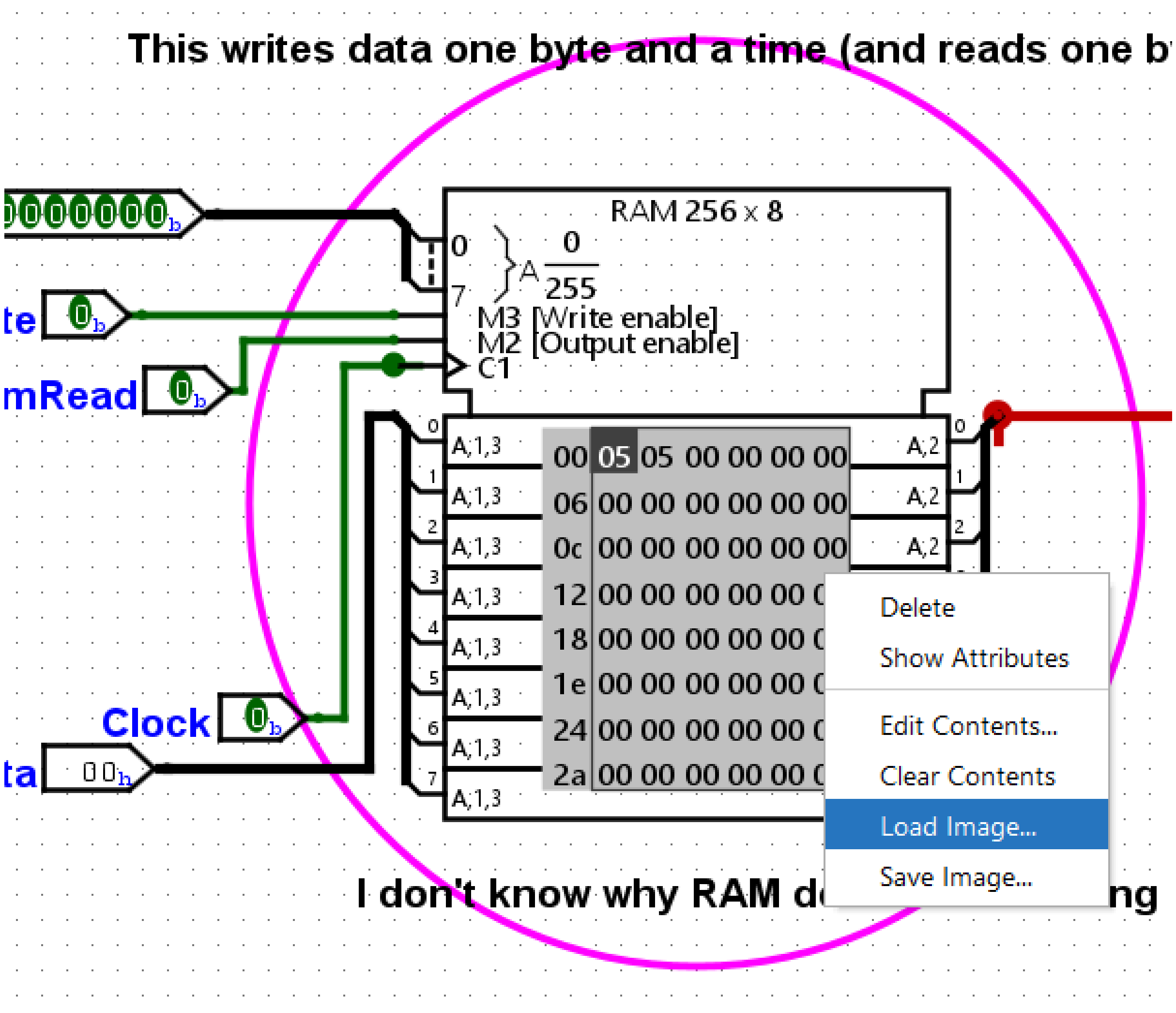


Now, go back to main.

Scroll down to DataMemory, click on it, and open the magnifying glass.

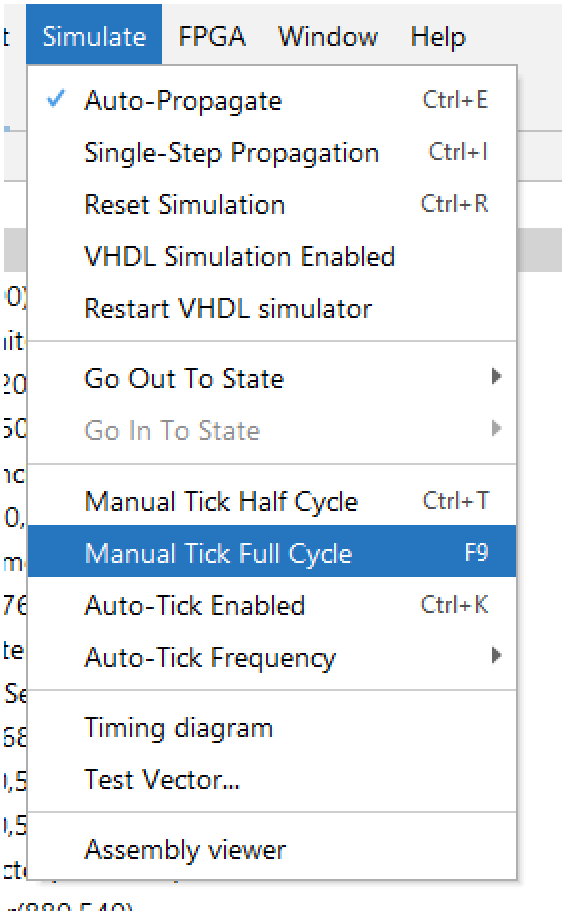


And load the Data image file.



You can open the DataMemory and InstructionMemory subcomponents anytime you want to view their values.

Now to run it, click on the simulate tab.



And run ticks.

You can do it manually (recommended) or enable auto tick.
Remember that if you enable Auto-tick, the instructions cut off at 255.