MICROPROGRAMMING
QUESTION 17

PRITISH WADHWA

- Program to find whether the value stored is perfect square or not:

```
/*Microcode Preamble*/
.begin:
        mloadIR
        mdecode
        madd        pc        ,4
        mswitch

mmovi   regData   ,1        <read>
mmov    regSrc    ,3        <write>
mmov    regSrc    ,2        <read>
mmov    sr1       ,regVal   <write>

mmovi   regSrc    ,3        <read>
mmov    B         ,regVal   <write>

.loop:
        mmov       A          ,B
        mmov       B          ,B         ,<mul>
        mmov       A          ,aluResult
        mmov       B          ,sr1
        mbeq       A          ,B         ,.perfect
        mmov       regSrc     ,3
        mmov       A          ,regVal
        mmovi      B          ,1         <write>
        mmov       A          ,A         ,<add>
        mmov       aluResult  ,sr1       ,<cmp>
        flags.GT   ,1         .notperfect

.perfect:
        mmov       regSrc     ,0         <read>
        mmov       regData    ,1         <write>
        mb         .begin

.notperfect:
        mmov       regSrc     ,0         <read>
        mmov       regData    ,0         <write>
        mb         .begin
```

- Explanation:

  - If the number is 0, we directly save the result as true
  - If the number is 1, we directly save the result as true
  - If the number is not 0 or 1, we branch to a loop
  - In the loop, we are aided by a variable 'B' which is initialized as 2.
  - We compute the square of 'B' and compare it with the test value.
  - If the square of B is greater than the test value, we save the result as false and exit from the loop and end the program.
  - If the square of B is equal to the test value, we save the result as true and exit from the loop and end the program.
  - If the square of B is less than the test value, we increment B by one and continue with the loop.
  - We keep doing the above steps until the program ends by any of the above cases.
  - The result is stored in the register R0.

MICROPROGRAMMING
QUESTION 18

PRITISH WADHWA

- Program to find whether the given number is Palindrome or not:

```
/*Microcode Preamble*/
.begin:
        mloadIR
        mdecode
        madd pc, 4
        mswitch

mmovi regSrc, 2, <read>
mmov A, regVal
mmov sr1, regVal
mmovi sr2, 3
mmovi sr5, 0

.loop:
        mmovi B, 1
        mmov A, A, <and>
        mmov sr3, aluResult

        mbeq sr3, 1, .store1

        .store0:
                mmov A, sr5
                mmovi B, 1
                mmov A, A, <lsl>
                mmov sr5, aluResult
                mb .shift

        .store1:
                mmov A, sr5
                mmovi B, 1
                mmov A, A, <lsl>
                mmov A, aluResult
                madd A, 1
                mmov sr5, A

        .shift:
```

```
        mmov A, sr1
        mmov B, 1
        mmov A, A, <lsr>
        mmov A, aluResult
        mmov sr1, A

mmov B, sr2
mbeq B, 0, .compare

madd B, -1
mmov sr2, B

mb .loop

.compare:
        mmov A, sr1
        mmov B, sr5
        mmov A, A, <cmp>

        mbeq flags.E, 1, .true
        mb .false

.true:
        mmov regSrc, 0, <read>
        mmov regData, 1, <write>
        mb .end

.false:
        mmov regSrc, 0, <read>
        mmov regData, 0, <write>
        mb .end

.begin:
```

Explanation:

- sr2 stores the count for the code
- last 4 bits of the 8-bit number is inverted and stored in sr5
- this is done by extracting the lsb from the main number and storing it in r5.
- sr1 meanwhile contains the whole number
- now sr2 is iterated till 4
- In each iteration the first bit is extracted from both sr1 and sr5.
- These bits are compared with each other.
- If the bits are not equal, the result is saved as false and the loop terminates and the code ends.
- If the bits are equal and the counter has exceeded its limit, the loop terminates, the result is saved as true and the code ends.
- If the counter(sr2) is less than 4, and the bits are equal, the loop continues.
- The code is run until any of the above statement becomes true.
- The final result is stored in R0.

PIPELINE DESIGN

Exercise 20

PRITISH WADHWA

- PART A

- [1]: **MUL** R8 ,R9 ,R10
  [2]: **ADD** R1 ,R2 ,R3
  [3]: *NOP*
  [4]: *NOP*
  [5]: **CMP** R8 ,R9
  [6]: **BEQ** .FOO
  [7]: **SUB** R4 ,R1 ,R1
  [8]: *NOP*

- In the above code, the statements are reordered such that the functionality remains the same and the pipeline functions in a way that is most efficient. The NOP instructions make sure that the input received to the consumer instructions is the correct input which is produced by the producer instructions.

- PART B

- [1]: **ADD** R1 ,R2 ,R3
  [2]: **MUL** R8 ,R9 ,R10
  [3]: **SUB** R4 ,R1 ,R1
  [4]: **CMP** R8 ,R9
  [5]: **BEQ** .FOO

	I	***II***	***III***	***IV***	***V***	***VI***	***VII***	***VIII***	***IX***	***X***	***XI***
IF	1	2	3	4	4	4	5	-	-	-	-
OF	-	1	2	3	3	3	4	5	-	-	-
EX	-	-	1	2	⦿	⦿	3	4	5	-	-
MA	-	-	-	1	2	⦿	⦿	3	4	5	-
RW	-	-	-	-	1	2	⦿	⦿	3	4	5

- ⦿ ➔ Bubble
- The bubbles in the above table make sure that the consumer instructions receive the correct and the intended input as generated by the producer instruction.

- PART C

- [1]: **MUL** R8 ,R9 ,R10
  [2]: **ADD** R1 ,R2 ,R3
  [3]: **SUB** R4 ,R1 ,R1
  [4]: **CMP** R8 ,R9
  [5]: **BEQ** .FOO

-

	I	II	III	IV	V	VI	VII	VIII	IX
IF	1	2	3	4	5	-	-	-	-
OF	-	1	2	3	4	5	-	-	-
EX	-	-	1	2	3	4	5	-	-
MA	-	-	-	1	2	3	4	5	-
RW	-	-	-	-	1	2	3	4	5

MA-EX Forwarding

RW-OF Forwarding

- Forwarding is a hardware method which makes sure that the pipeline becomes more efficient.
- It makes away with the use of Bubbles.
- It works in the principle that if correct value is available in some stage forward it to the stage which requires it.
- In the above question we use 3 types of forwarding:
    - MA-EX Forwarding
    - RW-OF Forwarding

PIPELINE DESIGN
Exercise 21

PRITISH WADHWA

- PART A:

- ```
  [1]:  ADD   R4   ,R3   ,R3
  [2]:  MUL   R8   ,R9   ,R10
  [3]:  DIV   R8   ,R9   ,R10
  [4]:  NOP
  [5]:  ST    R3   ,10[R4]
  [6]:  LD    R2   ,10[R4]
  [7]:  NOP
  [8]:  NOP
  [9]:  NOP
  [10]: ADD   R4   ,R2   ,R6
  ```

- In the above code, the statements are reordered such that the functionality remains the same and the pipeline functions in a way that is most efficient. The NOP instructions make sure that the input received to the consumer instructions is the correct input which is produced by the producer instructions.

- PART B:

- ```
  [1]:  ADD   R4   ,R3   ,R3
  [2]:  MUL   R8   ,R9   ,R10
  [3]:  DIV   R8   ,R9   ,R10
  [4]:  ST    R3   ,10[R4]
  [5]:  LD    R2   ,10[R4]
  [6]:  ADD   R4   ,R2   ,R6
  ```

	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII	XIV
IF	1	2	3	4	5	5	6	-	-	-	-	-	-	-
OF	-	1	2	3	4	4	5	6	6	6	6	-	-	-
EX	-	-	1	2	3	⊙	4	5	⊙	⊙	⊙	6	-	-
MA	-	-	-	1	2	3	⊙	4	5	⊙	⊙	⊙	6	-
RW	-	-	-	-	1	2	3	⊙	4	5	⊙	⊙	⊙	6

- ⊙ ➔ Bubble

- The bubbles in the above table make sure that the consumer instructions receive the correct and the intended input as generated by the producer instruction.

- PART C:

- [1]: **ADD** R4 ,R3 ,R3
  [2]: ST R3 ,10[R4]
  [3]: LD R2 ,10[R4]
  [4]: **DIV** R8 ,R9 ,R10
  [5]: **MUL** R8 ,R9 ,R10
  [6]: **ADD** R4 ,R2 ,R6

	I	II	III	IV	V	VI	VII	VIII	IX	X
IF	1	2	3	4	5	6	-	-	-	-
OF	-	1	2	3	4	5	6	-	-	-
EX	-	-	1	2	3	4	5	6	-	-
MA	-	-	-	1	2	3	4	5	6	-
RW	-	-	-	-	1	2	3	4	5	6

MA-EX Forwarding

RW-EX Forwarding

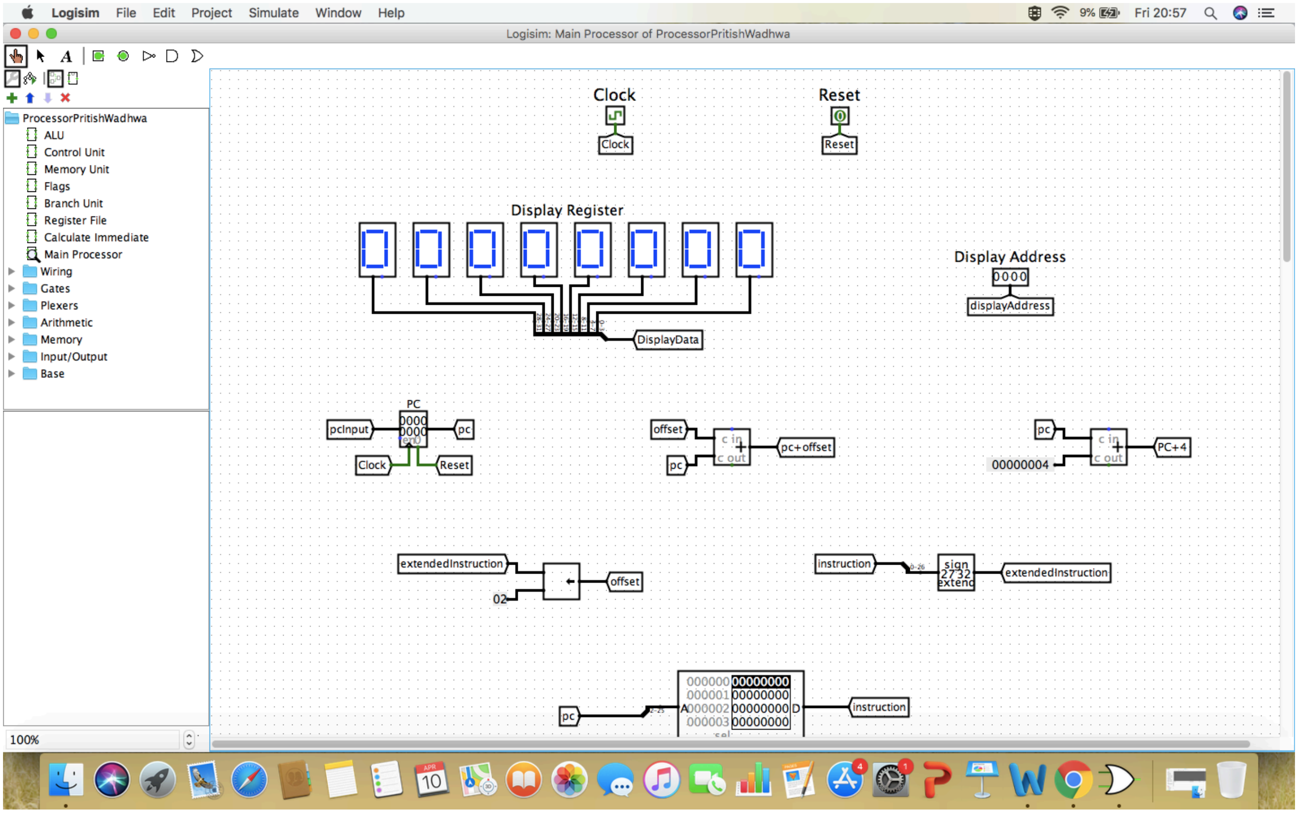
RW-OF Forwarding

- Forwarding is a hardware method which makes sure that the pipeline becomes more efficient.
- It makes away with the use of Bubbles.
- It works in the principle that if correct value is available in some stage forward it to the stage which requires it.
- In the above question we use 3 types of forwarding:
  - MA-EX Forwarding
  - RW-EX Forwarding
  - RW-OF Forwarding

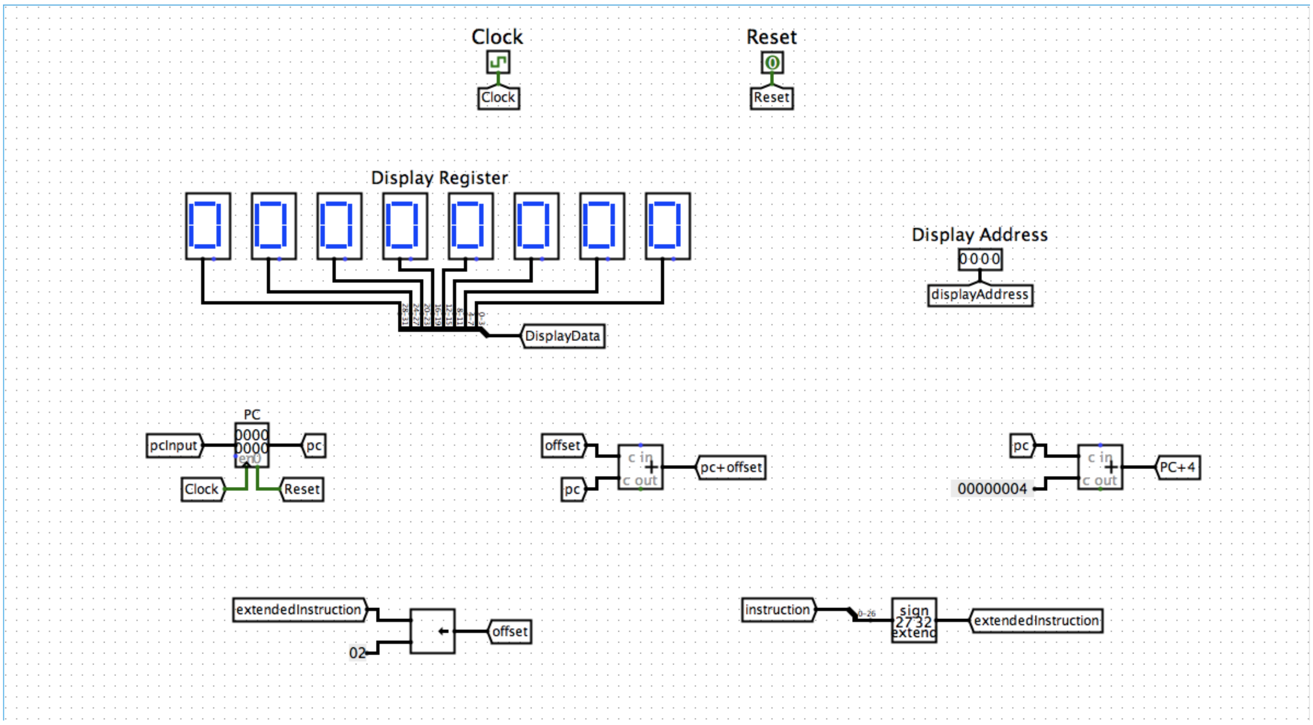
Bonus Task

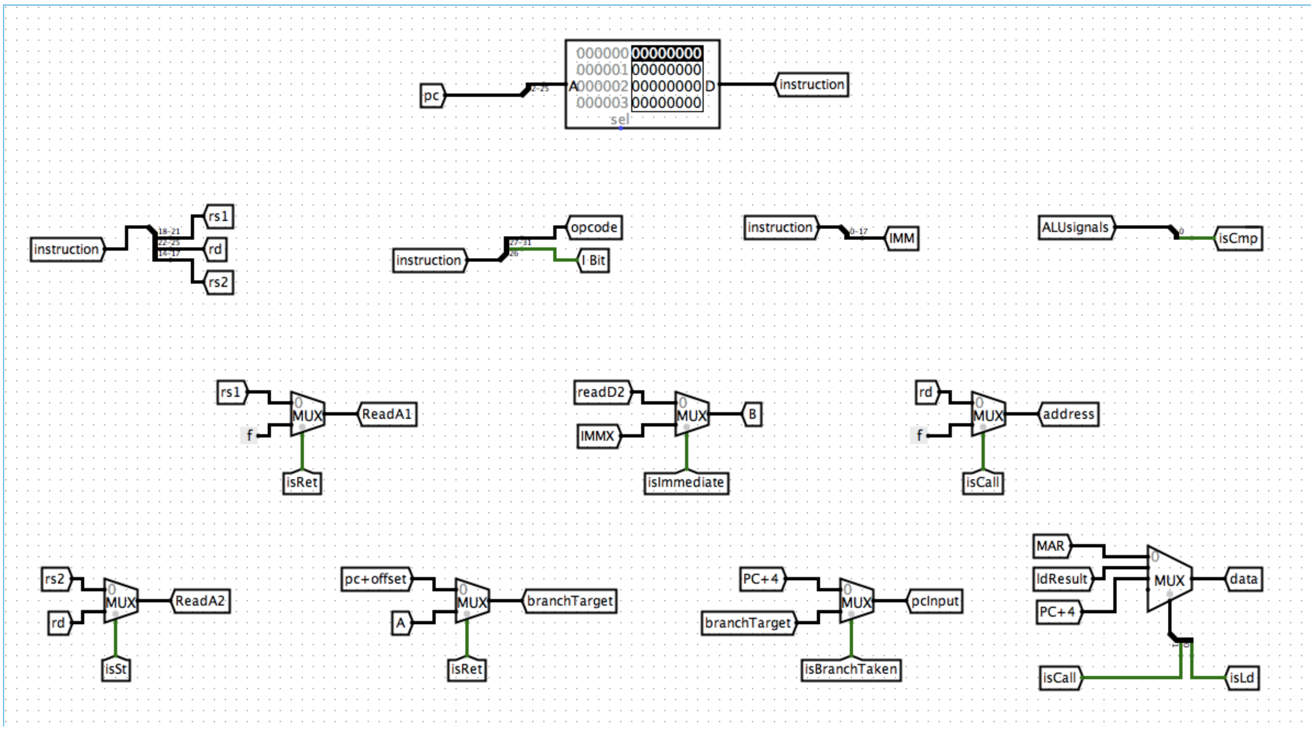
PRITISH WADHWA

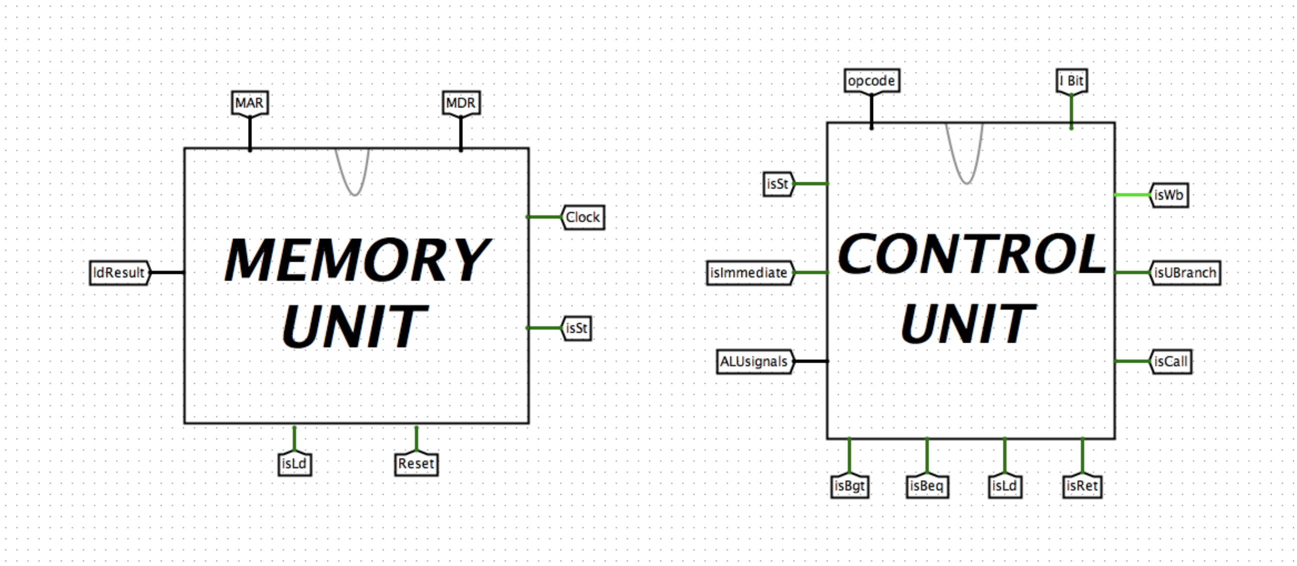
- Question 24
- Hardwired SimpleRisc processor using LogiSim
- Screenshots:
- Main Window:

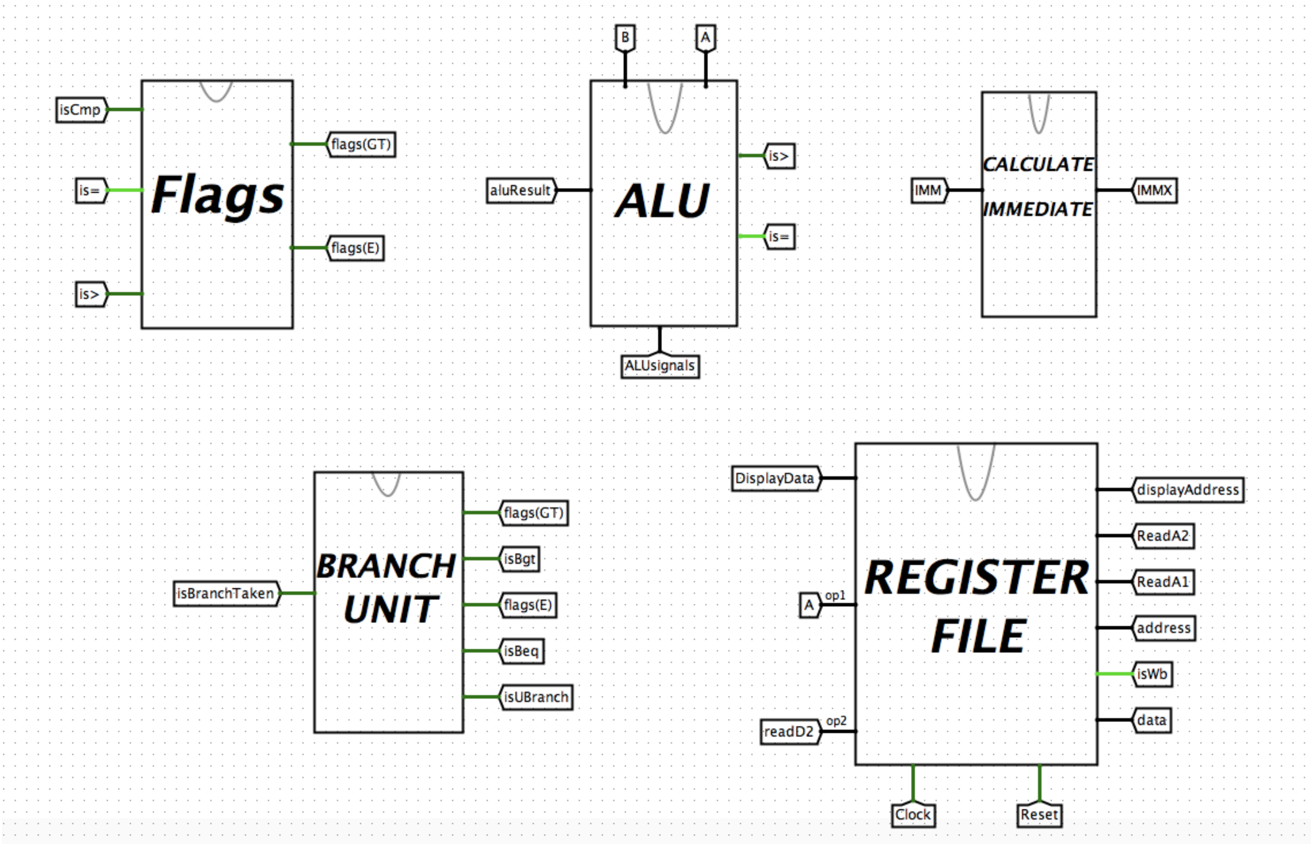


Main Processor:

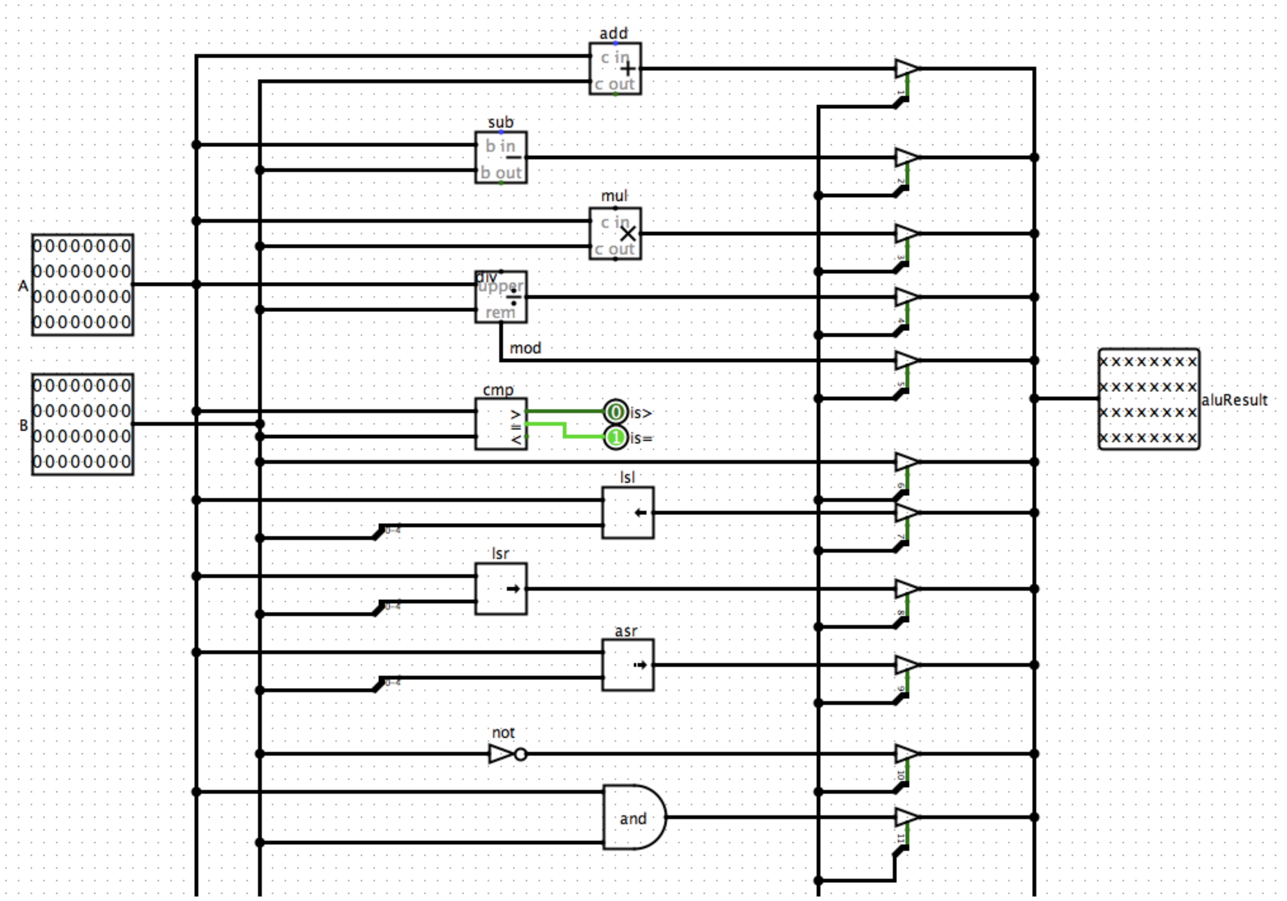


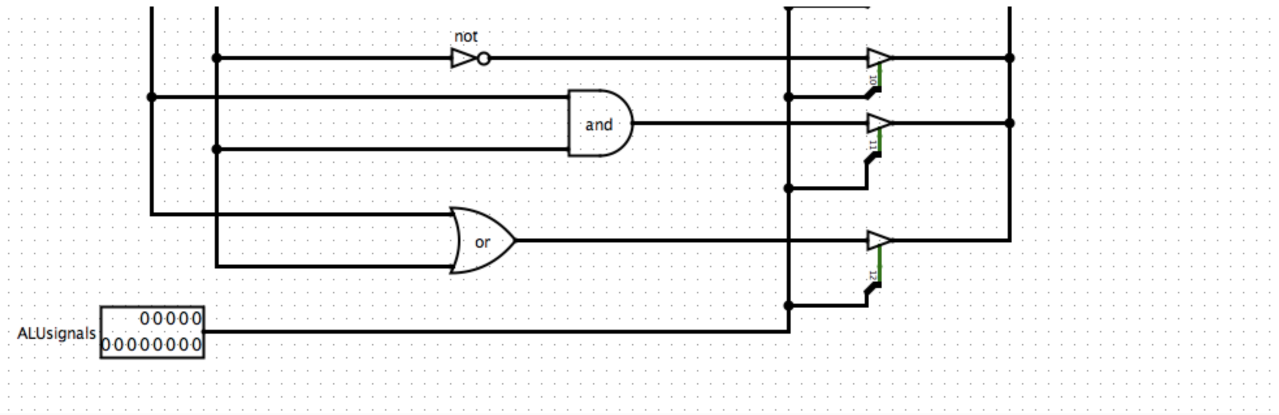



MAR
MDR
Clock
ldResult
MEMORY UNIT
isSt
isLd
Reset
opcode
I Bit
isSt
isWb
CONTROL UNIT
isImmediate
isUBranch
ALUsignals
isCall
isBgt
isBeq
isLd
isRet

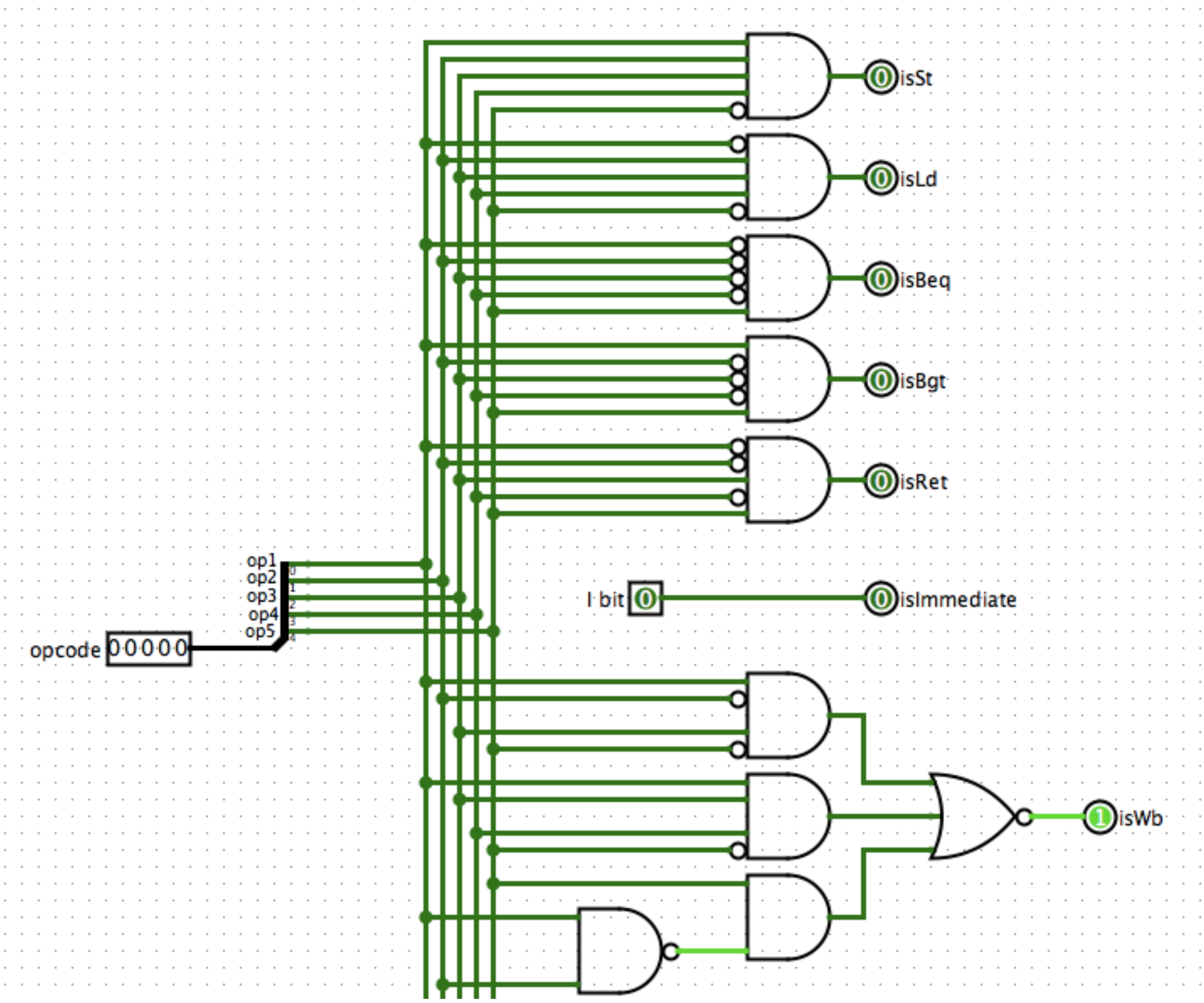

isCmp
Flags
flags(GT)
is=
flags(E)
is>
B
A
aluResult
ALU
is>
is=
ALUsignals
IMM
CALCULATE IMMEDIATE
IMMX
BRANCH UNIT
isBranchTaken
flags(GT)
isBgt
flags(E)
isBeq
isUBranch
DisplayData
REGISTER FILE
displayAddress
ReadA2
ReadA1
A
op1
address
isWb
readD2
op2
data
Clock
Reset

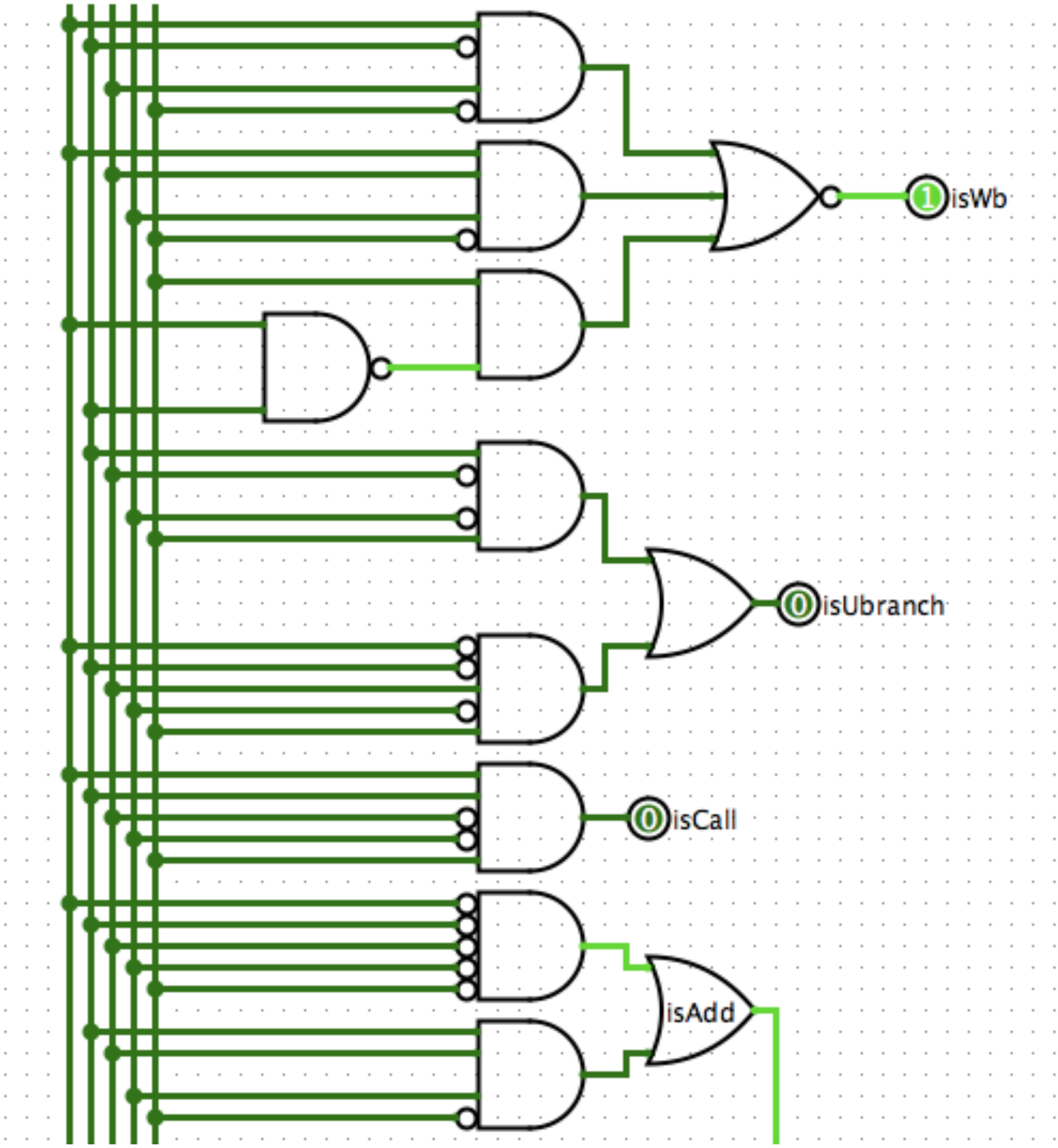
- ALU:


add
c in
c out
sub
b in
b out
mul
c in
c out
A
00000000
00000000
00000000
00000000
div
upper
rem
mod
B
00000000
00000000
00000000
00000000
cmp
is>
is=
lsl
lsr
asr
not
and
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
aluResult

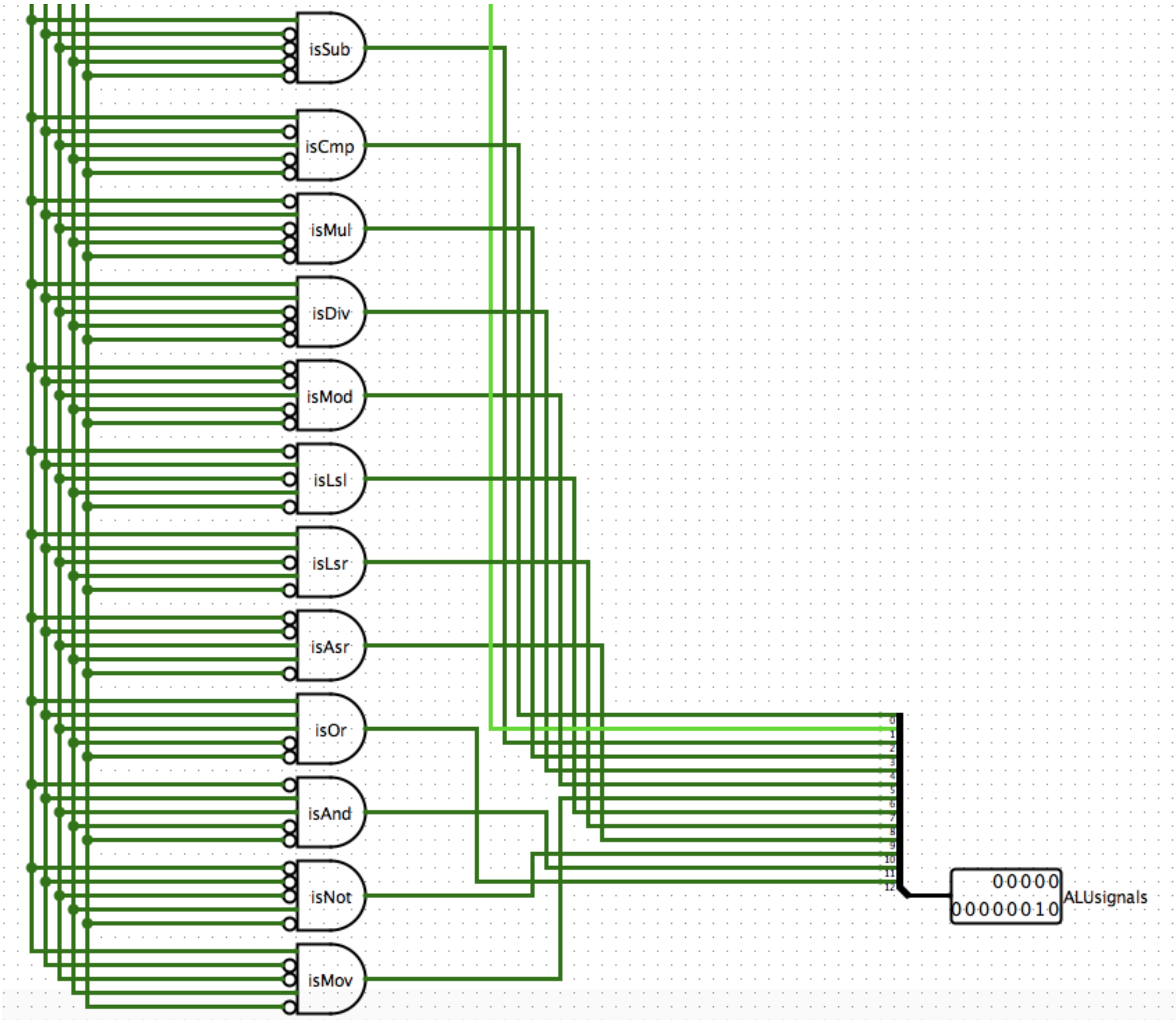

not
and
or
ALUsignals
00000
0000000

Control Unit:



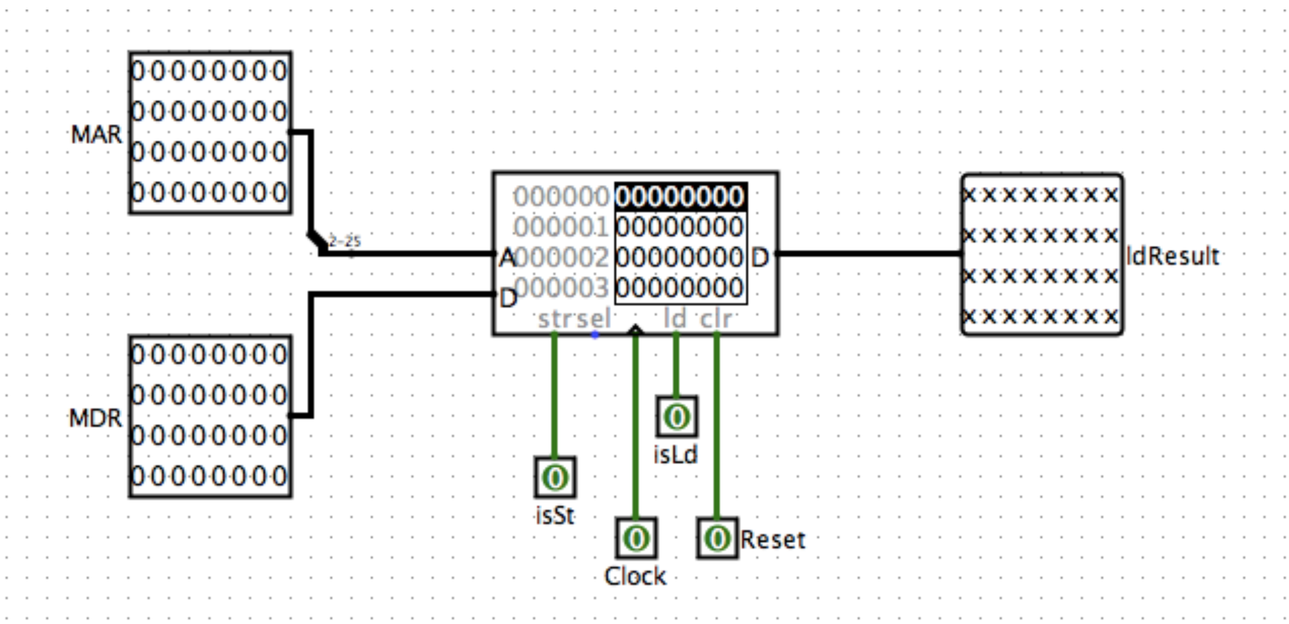


1
isWb
0
isUbranch
0
isCall
isAdd

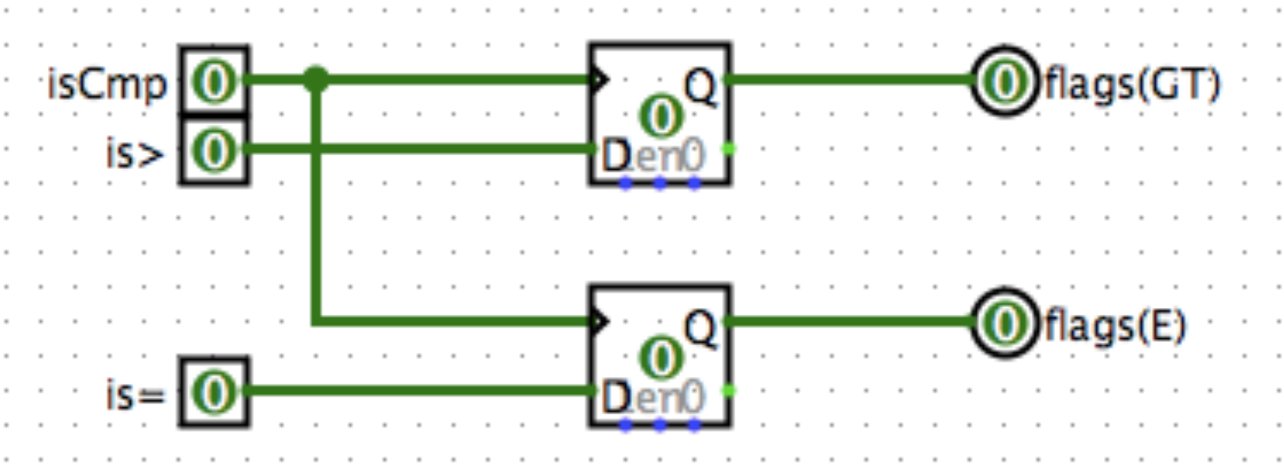


isSub
isCmp
isMul
isDiv
isMod
isLsl
isLsr
isAsr
isOr
isAnd
isNot
isMov
00000
00000010
ALUsignals

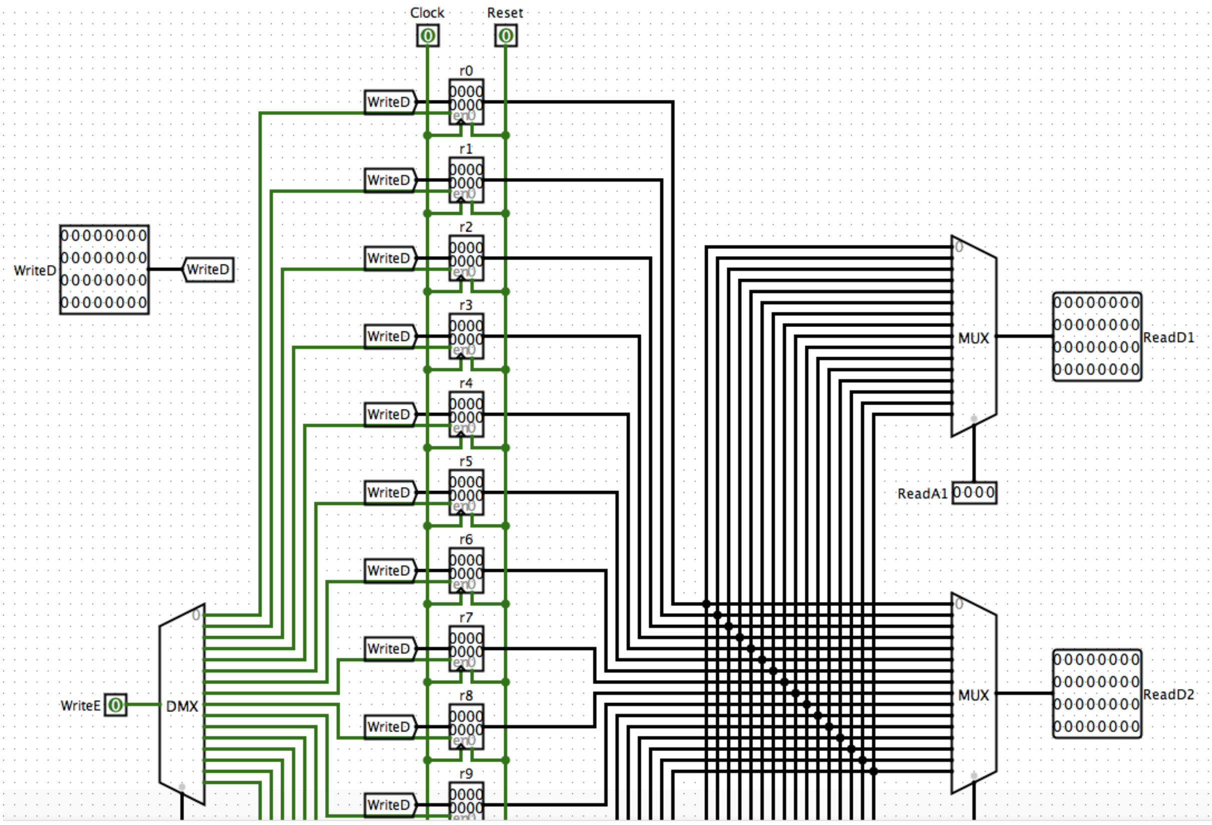
Memory Unit:

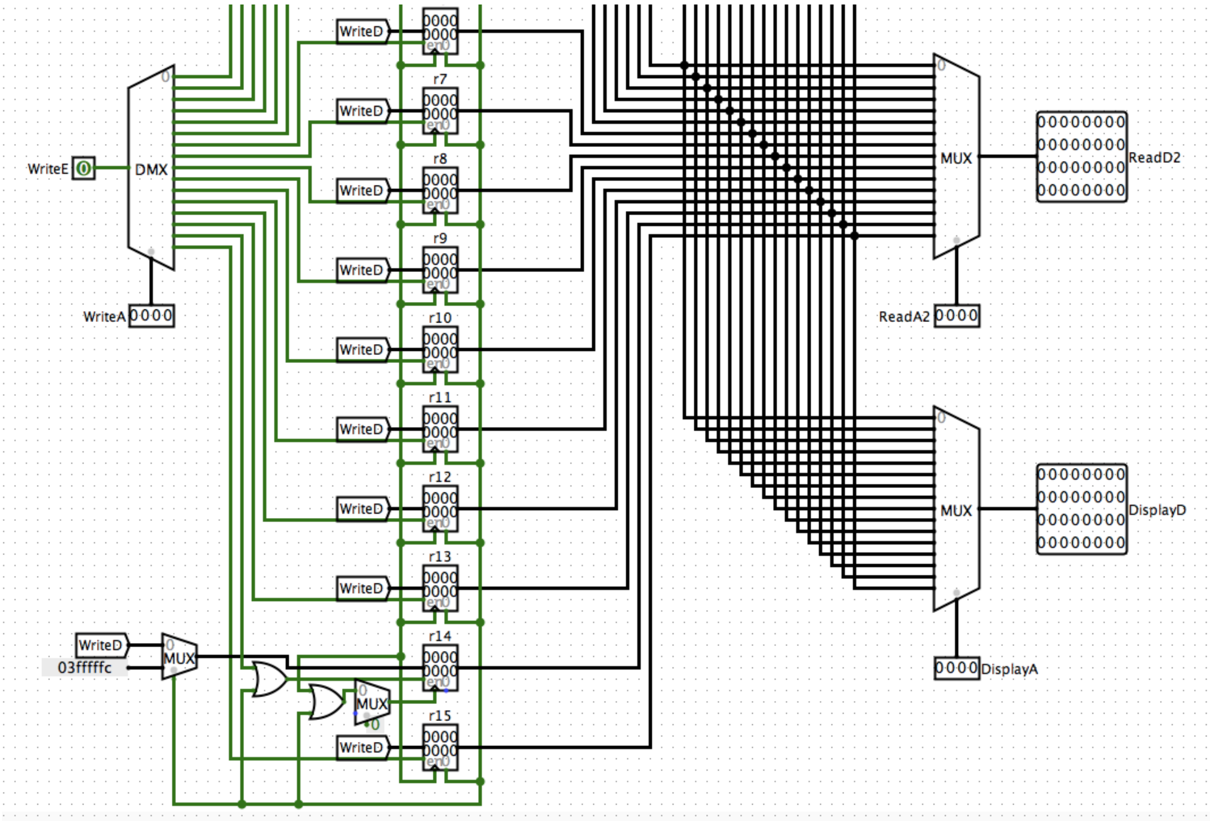


Flags:

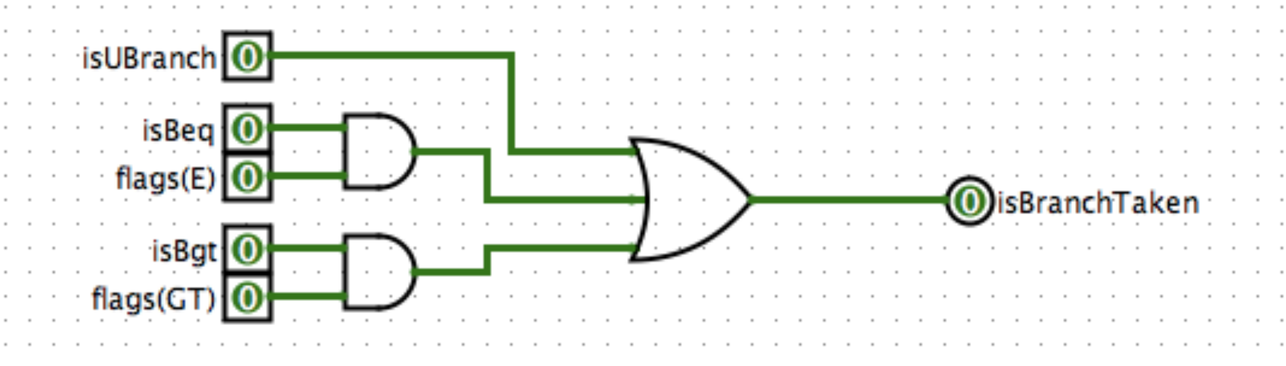


- Register File:





Branch Unit:



Calculate Immediate: