Design of a General-Purpose Microprocessor Using Software and Hardware Components

Phase 1

EE577A - Final Project

Samuel Bruner
Ming Hsieh Department of Electrical Engineering
University of Southern California
sbruner@usc.edu

Yulong Ding
Ming Hsieh Department of Electrical Engineering
University of Southern California
yulongdi@usc.edu

Angsagan Abdigazy
Ming Hsieh Department of Electrical Engineering
University of Southern California
abdigazy@usc.edu

Team 5
Yulong Ding 8456815156 - Front-end / Back-end Python Scripting
Samuel Bruner 3198542984 - Register File / System Integration
Angsagan Abdigazy 2509824628 - ALU

1. Python Scripting and Results

The Python script **project.py** combines the front-end instruction fetching/decoding and the back-end execution result verification in one script.

- **Front-end: Instruction Fetch and Decode**

  The front-end of the script takes a text file **sample_instructions.txt** as input, which contains the instruction sequence to be executed in an assembly-style syntax (as given on the DEN Course Final Project page), and produces vector files that assert appropriate control signals with proper timing for simulating the processor hardware design in Cadence. Read-after-write data dependencies that arise within the given instruction sequence when executed by the 5-stage pipelined processor are detected by the front-end, and appropriate numbers of "no operation" instructions (NOP) are then inserted between dependent instructions to resolve the detected hazard and ensure correct execution results. Because the SRAM memory block of the processor requires asynchronous control signals to perform load and store operations, the front-end also generates a separate vector file **mem.vec** to control precharging, reading from and writing to the SRAM. All other control signals, data and address inputs are generated in the main vector file **cpu.vec**.

```python
# Front-end: Instruction Fetch and Decode
fileIn = open('sample_instructions.txt', 'r')
fileOutCPU = open('cpu.vec', 'w')
fileOutMEM = open('mem.vec', 'w')
fileOutCPU.write('radix 1 1 1 3 3 3 ')
fileOutCPU.write('1 1 1 1 1 1 1 ')
fileOutCPU.write('1 1 1 4444 14\n')
fileOutCPU.write('io i i i i i i i i i i i i i i i i i i\n')
fileOutCPU.write('vname clk ~reset reg_write reg_dst<[2:0]> reg_src_1<[2:0]> reg_src_2<[2:0]> ')
fileOutCPU.write('and_en or_en add_en mul_en min_en shift_en right_shift ')
fileOutCPU.write('imm_op alu_load_sel loadi_sel data_in<[15:0]> addr<[4:0]>\n')
fileOutCPU.write('tunit ns\nslope 0.005\nvih 1.0\nvil 0.0\n\n')
fileOutMEM.write('radix 1 1 1 1\n')
fileOutMEM.write('io i i i i\n')
fileOutMEM.write('vname ~pre_en mem_write_en mem_read_en rowdec_en\n')
fileOutMEM.write('tunit ns\nslope 0.005\nvih 1.0\nvil 0.0\n\n')
destination = ['none', 'none', 'none']
instructionList = []
commandNum = 0
for line in fileIn:
    line = line.replace('$', '')
    line = line.replace('#', '')
    line = line.upper()
    instruction = line.split(' ')
    if instruction == ['\n']:
        continue
    commandNum += 1
    instruction[-1] = instruction[-1].replace('\n', '')
    if instruction[0] == 'STOREI' or instruction[0] == 'STORE' or instruction[0] == 'LOAD':
        for i in range(1, 3):
            if len(instruction[i]) == 3 and instruction[i][-1] == 'H':
                instruction[i] = instruction[i].replace('H', '')
            elif len(instruction[i]) == 6 and instruction[i][-1] == 'B':
                instruction[i] = '%X' % int(instruction[i].replace('B', ''), 2)
                if len(instruction[i]) == 1:
                    instruction[i] = '0' + instruction[i]
    if instruction[0] == 'STOREI':
        if len(instruction[1]) == 1:
            addrLSB = instruction[2][-1]
            if instruction[1] == '1':
                instruction.pop(1)
                instructionList.append(instruction)
                destination.pop(0)
                destination.append('none')
            elif instruction[1] == '2':
                if (addrLSB != '0' and addrLSB != '2' and addrLSB != '4' and addrLSB != '6'
                    and addrLSB != '8' and addrLSB != 'A' and addrLSB != 'C' and addrLSB != 'E'):
                    print('Error001: Command ' + str(commandNum) + ' is not aligned properly.')
                    continue
                else:
                    instructionList.append(['STOREI', instruction[2], instruction[3]])
                    instructionList.append(['STOREI', instruction[2][0] + chr(ord(addrLSB) + 1), instruction[4]])
                    destination.pop(0)
                    destination.pop(0)
                    destination.append('none')
                    destination.append('none')
            elif instruction[1] == '4':
                if addrLSB != '0' and addrLSB != '4' and addrLSB != '8' and addrLSB != 'C':
                    print('Error001: Command ' + str(commandNum) + ' is not aligned properly.')
                    continue
                else:
                    instructionList.append(['STOREI', instruction[2], instruction[3]])
                    instructionList.append(['STOREI', instruction[2][0] + chr(ord(addrLSB) + 1), instruction[4]])
                    if addrLSB == '8':
                        instructionList.append(['STOREI', instruction[2][0] + 'A', instruction[5]])
                        instructionList.append(['STOREI', instruction[2][0] + 'B', instruction[6]])
```

```
                    else:
                        instructionList.append(['STOREI', instruction[2][0] + chr(ord(addrLSB) + 2), instruction[5]])
                        instructionList.append(['STOREI', instruction[2][0] + chr(ord(addrLSB) + 3), instruction[6]])
                    destination = ['none', 'none', 'none']
            else:
                print('Error000: Command ' + str(commandNum) + ' has invalid burst length.')
                continue
        else:
            instructionList.append(instruction)
            destination.pop(0)
            destination.append('none')
    else:
        if instruction[0] != 'LOADI' and instruction[0] != 'LOAD' and instruction[0] != 'NOP':
            while instruction[2] in destination or (instruction[0] != 'STORE' and instruction[3] in destination):
                instructionList.append(['NOP'])
                destination.pop(0)
                destination.append('none')
        instructionList.append(instruction)
        destination.pop(0)
        if instruction[0] != 'STORE' and instruction[0] != 'NOP':
            destination.append(instruction[1])
        else:
            destination.append('none')
clkPeriod = 1
clk = '1'
reset_bar = '0'
reg_write = '0'
reg_dst = '0'
reg_src_1 = '0'
reg_src_2 = '0'
and_en = '0'
or_en = '0'
add_en = '0'
mul_en = '0'
min_en = '0'
shift_en = '0'
right_shift = '0'
imm_op = '0'
alu_load_sel = '0'
loadi_sel = '0'
data_in = '0000'
addr = '00'
fileOutCPU.write('0    ')
fileOutCPU.write(clk + ' ' + reset_bar + ' ' + reg_write + ' ' + reg_dst + ' ' + reg_src_1 + ' ' + reg_src_2 + ' ')
fileOutCPU.write(and_en + ' ' + or_en + ' ' + add_en + ' ' + mul_en + ' ' + min_en + ' ' + shift_en + ' ' + right_shift + ' ')
fileOutCPU.write(imm_op + ' ' + alu_load_sel + ' ' + loadi_sel + ' ' + data_in + ' ' + addr + '\n')
fileOutMEM.write('0  1 0 0 0\n')
reset_bar = '1'
for i in range(len(instructionList)):
    op = instructionList[i]
    if op[0] == 'STOREI' or op[0] == 'STORE' or op[0] == 'NOP':
        reg_write = '0'
    else:
        reg_write = '1'
        reg_dst = op[1]
    if op[0] != 'STOREI' and op[0] != 'STORE' and op[0] != 'LOADI' and op[0] != 'LOAD' and op[0] != 'NOP':
        reg_src_1 = op[2]
    if op[0] == 'STORE':
        reg_src_2 = op[2]
    elif op[0] == 'AND' or op[0] == 'OR' or op[0] == 'ADD' or op[0] == 'MUL' or op[0] == 'MIN':
        reg_src_2 = op[3]
    if op[0] == 'ANDI' or op[0] == 'AND':
        and_en = '1'
    else:
        and_en = '0'
    if op[0] == 'ORI' or op[0] == 'OR':
```

```
        or_en = '1'
    else:
        or_en = '0'
    if op[0] == 'ADDI' or op[0] == 'MINI' or op[0] == 'ADD' or op[0] == 'MIN':
        add_en = '1'
    else:
        add_en = '0'
    if op[0] == 'MULI' or op[0] == 'MUL':
        mul_en = '1'
    else:
        mul_en = '0'
    if op[0] == 'MINI' or op[0] == 'MIN':
        min_en = '1'
    else:
        min_en = '0'
    if op[0] == 'SFL':
        shift_en = '1'
        right_shift = '0'
    elif op[0] == 'SFR':
        shift_en = '1'
        right_shift = '1'
    else:
        shift_en = '0'
    if op[0] == 'STORE' or op[0] == 'LOAD' or op[0] == 'AND' or op[0] == 'OR' or op[0] == 'ADD' or op[0] == 'MUL' or op[0] == 'MIN':
        imm_op = '0'
    else:
        imm_op = '1'
    if op[0] == 'LOAD':
        alu_load_sel = '1'
    else:
        alu_load_sel = '0'
    if op[0] == 'LOADI':
        loadi_sel = '1'
    else:
        loadi_sel = '0'
    if op[0] == 'STOREI' or op[0] == 'LOADI':
        data_in = op[2]
    elif op[0] == 'ANDI' or op[0] == 'ORI'or op[0] == 'ADDI' or op[0] == 'MINI' or op[0] == 'SFL' or op[0] == 'SFR':
        data_in = op[3]
    elif op[0] == 'MULI':
        data_in = '00' + op[3]
    if op[0] == 'STOREI' or op[0] == 'STORE':
        addr = op[1]
        fileOutMEM.write('\n; ')
        for argument in op:
            fileOutMEM.write(argument + ' ')
        fileOutMEM.write('\n')
        fileOutMEM.write(str(i * clkPeriod + 2 * clkPeriod) + '  ')
        fileOutMEM.write('0 0 0 0\n')
        fileOutMEM.write(str(i * clkPeriod + 2 * clkPeriod + 0.1) + '  ')
        fileOutMEM.write('1 1 0 1\n')
        fileOutMEM.write(str(i * clkPeriod + 2 * clkPeriod + clkPeriod * 0.8) + '  ')
        fileOutMEM.write('1 0 0 0\n')
    elif op[0] == 'LOAD':
        addr = op[2]
        fileOutMEM.write('\n; ')
        for argument in op:
            fileOutMEM.write(argument + ' ')
        fileOutMEM.write('\n')
        fileOutMEM.write(str(i * clkPeriod + 2 * clkPeriod) + '  ')
        fileOutMEM.write('0 0 0 0\n')
        fileOutMEM.write(str(i * clkPeriod + 2 * clkPeriod + 0.1) + '  ')
        fileOutMEM.write('1 0 0 1\n')
        fileOutMEM.write(str(i * clkPeriod + 2 * clkPeriod + clkPeriod * 0.6) + '  ')
        fileOutMEM.write('1 0 1 0\n')
        fileOutMEM.write(str(i * clkPeriod + 2 * clkPeriod + clkPeriod * 0.9) + '  ')
        fileOutMEM.write('1 0 0 0\n')
    fileOutCPU.write('\n; ')
    for argument in op:
        fileOutCPU.write(argument + ' ')
    fileOutCPU.write('\n')
    clk = '0'
    fileOutCPU.write(str(i * clkPeriod + clkPeriod / 2) + '  ')
    fileOutCPU.write(clk + ' ' + reset_bar + ' ' + reg_write + ' ' + reg_dst + ' ' + reg_src_1 + ' ' + reg_src_2 + ' ')
    fileOutCPU.write(and_en + ' ' + or_en + ' ' + add_en + ' ' + mul_en + ' ' + min_en + ' ' + shift_en + ' ' + right_shift + ' ')
    fileOutCPU.write(imm_op + ' ' + alu_load_sel + ' ' + loadi_sel + ' ' + data_in + ' ' + addr + '\n')
    clk = '1'
    fileOutCPU.write(str(i * clkPeriod + clkPeriod) + '    ')
    fileOutCPU.write(clk + ' ' + reset_bar + ' ' + reg_write + ' ' + reg_dst + ' ' + reg_src_1 + ' ' + reg_src_2 + ' ')
    fileOutCPU.write(and_en + ' ' + or_en + ' ' + add_en + ' ' + mul_en + ' ' + min_en + ' ' + shift_en + ' ' + right_shift + ' ')
    fileOutCPU.write(imm_op + ' ' + alu_load_sel + ' ' + loadi_sel + ' ' + data_in + ' ' + addr + '\n')
fileOutCPU.write('\n; END OF INSTRUCTION SEQUENCE')
for j in range(1, 6):
    clk = '0'
    fileOutCPU.write('\n' + str((i + j) * clkPeriod + clkPeriod / 2) + '  ')
    fileOutCPU.write(clk + ' 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00\n')
    clk = '1'
    fileOutCPU.write(str((i + j) * clkPeriod + clkPeriod) + '    ')
    fileOutCPU.write(clk + ' 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00\n')
fileIn.close()
fileOutCPU.close()
fileOutMEM.close()
```

- **Back-end: Automated Result Verification**

  The back-end of the script simulates the execution result of the given sequence by the pipelined processor and produces a text file **golden_results.txt** to show the final contents of the Register File and the SRAM after executing the entire sequence. By comparing the Cadence simulation results with the golden results generated by the back-end script, the correct functionality of the processor hardware design can be verified. Note that the back-end has been implemented to simulate exactly the behavior of the hardware, such as overflow in 16-bit signed addition, sign-extending the 10-bit product of the lower 5-bit signed multiplication, performing arithmetic (signed) bit shifts, etc.

```python
# Back-end: Automated Result Verification
def signed(value, bits):
    if value & (1 << (bits - 1)):
        value = value - (1 << bits)
    return value
def unsigned(value, bits):
    if value < 0:
        value = value + (1 << bits)
    return value
fileOutGolden = open('golden_results.txt', 'w')
SRAM = {}
RF = {}
for instruction in instructionList:
    if instruction[0] == 'STOREI':
        SRAM[instruction[1]] = int(instruction[2], 16)
    elif instruction[0] == 'LOADI':
        RF[instruction[1]] = int(instruction[2], 16)
    elif instruction[0] == 'STORE':
        if instruction[2] in RF.keys():
            SRAM[instruction[1]] = RF[instruction[2]]
    elif instruction[0] == 'LOAD':
        if instruction[2] in SRAM.keys():
            RF[instruction[1]] = SRAM[instruction[2]]
    elif instruction[0] == 'AND':
        if instruction[2] in RF.keys() and instruction[3] in RF.keys():
            RF[instruction[1]] = RF[instruction[2]] & RF[instruction[3]]
    elif instruction[0] == 'ANDI':
        if instruction[2] in RF.keys():
            RF[instruction[1]] = RF[instruction[2]] & int(instruction[3], 16)
    elif instruction[0] == 'OR':
        if instruction[2] in RF.keys() and instruction[3] in RF.keys():
            RF[instruction[1]] = RF[instruction[2]] | RF[instruction[3]]
    elif instruction[0] == 'ORI':
        if instruction[2] in RF.keys():
            RF[instruction[1]] = RF[instruction[2]] | int(instruction[3], 16)
    elif instruction[0] == 'ADD':
        if instruction[2] in RF.keys() and instruction[3] in RF.keys():
            result = RF[instruction[2]] + RF[instruction[3]]
            result_hex = hex(result).replace('0x', '')
            if len(result_hex) > 4:
                result_hex = result_hex[1:]
            result = int(result_hex, 16)
            RF[instruction[1]] = result
    elif instruction[0] == 'ADDI':
        if instruction[2] in RF.keys():
            result = RF[instruction[2]] + int(instruction[3], 16)
            result_hex = hex(result).replace('0x', '')
            if len(result_hex) > 4:
                result_hex = result_hex[1:]
            result = int(result_hex, 16)
            RF[instruction[1]] = result
    elif instruction[0] == 'MUL':
        if instruction[2] in RF.keys() and instruction[3] in RF.keys():
            op1 = signed(int(bin(RF[instruction[2]]).replace('0b', '')[-5:], 2), 5)
            op2 = signed(int(bin(RF[instruction[3]]).replace('0b', '')[-5:], 2), 5)
            result = op1 * op2
            RF[instruction[1]] = unsigned(result, 16)
    elif instruction[0] == 'MULI':
        if instruction[2] in RF.keys():
            op1 = signed(int(bin(RF[instruction[2]]).replace('0b', '')[-5:], 2), 5)
            op2 = signed(int(instruction[3], 16), 5)
            result = op1 * op2
            RF[instruction[1]] = unsigned(result, 16)
    elif instruction[0] == 'MIN':
        if instruction[2] in RF.keys() and instruction[3] in RF.keys():
            if signed(RF[instruction[2]], 16) < signed(RF[instruction[3]], 16):
                RF[instruction[1]] = RF[instruction[2]]
            else:
                RF[instruction[1]] = RF[instruction[3]]
    elif instruction[0] == 'MINI':
        if instruction[2] in RF.keys():
            if signed(RF[instruction[2]], 16) < signed(int(instruction[3], 16), 16):
                RF[instruction[1]] = RF[instruction[2]]
            else:
                RF[instruction[1]] = int(instruction[3], 16)
    elif instruction[0] == 'SFL':
        if instruction[2] in RF.keys():
            result = RF[instruction[2]] << int(instruction[3], 16)
            result_hex = hex(result).replace('0x', '')
            if len(result_hex) > 4:
                result_hex = result_hex[-4:]
                result = int(result_hex, 16)
            RF[instruction[1]] = result
    elif instruction[0] == 'SFR':
        if instruction[2] in RF.keys():
            RF[instruction[1]] = unsigned(signed(RF[instruction[2]], 16) >> int(instruction[3], 16), 16)
fileOutGolden.write('Register File - Final Content:\n')
for key in sorted(RF):
    fileOutGolden.write('$' + key + ': ')
    content_hex = hex(RF[key]).replace('0x', '')
    while len(content_hex) < 4:
        content_hex = '0' + content_hex
    fileOutGolden.write(content_hex.upper() + '\n')
fileOutGolden.write('\nSRAM - Final Content:\n')
for key in sorted(SRAM):
    fileOutGolden.write(key + 'H: ')
    content_hex = hex(SRAM[key]).replace('0x', '')
    while len(content_hex) < 4:
        content_hex = '0' + content_hex
    fileOutGolden.write(content_hex.upper() + '\n')
fileOutGolden.close()
```

- Front-end generated Cadence vector file: **cpu.vec**

```
radix 1 1 1 3 3 3 1 1 1 1 1 1 1 1 1 1 4444 14
io i i i i i i i i i i i i i i i i i i
vname clk ~reset reg_write reg_dst<[2:0]> reg_src_1<[2:0]> reg_src_2<[2:0]> and_en or_en add_en mul_en min_en shift_en right_shift imm_op alu_load_sel loadi_sel data_in<[15:0]> addr<[4:0]>
tunit ns
slope 0.005
vih 1.0
vil 0.0

0    1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00

; STOREI 0A 001F
0.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 001F 0A
1    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 001F 0A

; STOREI 0B 0002
1.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0002 0B
2    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0002 0B

; STOREI 10 000F
2.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 000F 10
3    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 000F 10

; STOREI 11 00FE
3.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 11
4    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 11

; LOAD 1 0A
4.5  0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 0A
5    1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 0A

; LOAD 2 0B
5.5  0 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 0B
6    1 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 0B

; LOAD 3 10
6.5  0 1 1 3 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 10
7    1 1 1 3 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 10

; LOAD 4 11
7.5  0 1 1 4 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 11
8    1 1 1 4 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 11

; NOP
8.5  0 1 0 4 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 11
9    1 1 0 4 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 00FE 11

; MUL 5 1 2
9.5  0 1 1 5 1 2 0 0 0 1 0 0 0 0 0 0 00FE 11
10   1 1 1 5 1 2 0 0 0 1 0 0 0 0 0 0 00FE 11

; NOP
10.5  0 1 0 5 1 2 0 0 0 0 0 0 0 0 1 0 0 00FE 11
11    1 1 0 5 1 2 0 0 0 0 0 0 0 0 1 0 0 00FE 11

; ADD 6 3 4
11.5  0 1 1 6 3 4 0 0 1 0 0 0 0 0 0 0 00FE 11
12    1 1 1 6 3 4 0 0 1 0 0 0 0 0 0 0 00FE 11

; NOP
12.5  0 1 0 6 3 4 0 0 0 0 0 0 0 0 1 0 0 00FE 11
13    1 1 0 6 3 4 0 0 0 0 0 0 0 0 1 0 0 00FE 11

; STORE 00 5
13.5  0 1 0 6 3 5 0 0 0 0 0 0 0 0 0 0 00FE 00
14    1 1 0 6 3 5 0 0 0 0 0 0 0 0 0 0 00FE 00

```

```
; NOP
14.5  0 1 0 6 3 5 0 0 0 0 0 0 0 1 0 0 00FE 00
15    1 1 0 6 3 5 0 0 0 0 0 0 0 1 0 0 00FE 00

; STORE 01 6
15.5  0 1 0 6 3 6 0 0 0 0 0 0 0 0 0 0 00FE 01
16    1 1 0 6 3 6 0 0 0 0 0 0 0 0 0 0 00FE 01

; SFL 5 3 0002
16.5  0 1 1 5 3 6 0 0 0 0 0 1 0 1 0 0 0002 01
17    1 1 1 5 3 6 0 0 0 0 0 1 0 1 0 0 0002 01

; OR 6 2 4
17.5  0 1 1 6 2 4 0 1 0 0 0 0 0 0 0 0 0002 01
18    1 1 1 6 2 4 0 1 0 0 0 0 0 0 0 0 0002 01

; NOP
18.5  0 1 0 6 2 4 0 0 0 0 0 0 0 1 0 0 0002 01
19    1 1 0 6 2 4 0 0 0 0 0 0 0 1 0 0 0002 01

; NOP
19.5  0 1 0 6 2 4 0 0 0 0 0 0 0 1 0 0 0002 01
20    1 1 0 6 2 4 0 0 0 0 0 0 0 1 0 0 0002 01

; AND 7 5 3
20.5  0 1 1 7 5 3 1 0 0 0 0 0 0 0 0 0 0002 01
21    1 1 1 7 5 3 1 0 0 0 0 0 0 0 0 0 0002 01

; STORE 02 5
21.5  0 1 0 7 5 5 0 0 0 0 0 0 0 0 0 0 0002 02
22    1 1 0 7 5 5 0 0 0 0 0 0 0 0 0 0 0002 02

; STORE 03 6
22.5  0 1 0 7 5 6 0 0 0 0 0 0 0 0 0 0 0002 03
23    1 1 0 7 5 6 0 0 0 0 0 0 0 0 0 0 0002 03

; NOP
23.5  0 1 0 7 5 6 0 0 0 0 0 0 0 1 0 0 0002 03
24    1 1 0 7 5 6 0 0 0 0 0 0 0 1 0 0 0002 03

; STORE 04 7
24.5  0 1 0 7 5 7 0 0 0 0 0 0 0 0 0 0 0002 04
25    1 1 0 7 5 7 0 0 0 0 0 0 0 0 0 0 0002 04

; LOAD 0 00
25.5  0 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 00
26    1 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 00

; LOAD 0 01
26.5  0 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 01
27    1 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 01

; LOAD 0 02
27.5  0 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 02
28    1 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 02

; LOAD 0 03
28.5  0 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 03
29    1 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 03

; LOAD 0 04
29.5  0 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 04
30    1 1 1 0 5 7 0 0 0 0 0 0 0 0 1 0 0002 04

; END OF INSTRUCTION SEQUENCE
30.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00
31    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00

31.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00
32    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00

32.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00
33    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00

33.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00
34    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00

34.5  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00
35    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0000 00
```

- Front-end generated Cadence vector file: **mem.vec**

```
radix 1 1 1 1
io i i i i
vname ~pre_en mem_write_en mem_read_en rowdec_en
tunit ns
slope 0.005
vih 1.0
vil 0.0

0  1 0 0 0

; STOREI 0A 001F
2  0 0 0 0
2.1  1 1 0 1
2.8  1 0 0 0

; STOREI 0B 0002
3  0 0 0 0
3.1  1 1 0 1
3.8  1 0 0 0

; STOREI 10 000F
4  0 0 0 0
4.1  1 1 0 1
4.8  1 0 0 0

; STOREI 11 00FE
5  0 0 0 0
5.1  1 1 0 1
5.8  1 0 0 0

; LOAD 1 0A
6  0 0 0 0
6.1  1 0 0 1
6.6  1 0 1 0
6.9  1 0 0 0

; LOAD 2 0B
7  0 0 0 0
7.1  1 0 0 1
7.6  1 0 1 0
7.9  1 0 0 0

; LOAD 3 10
8  0 0 0 0
8.1  1 0 0 1
8.6  1 0 1 0
8.9  1 0 0 0

; LOAD 4 11
9  0 0 0 0
9.1  1 0 0 1
9.6  1 0 1 0
9.9  1 0 0 0

; STORE 00 5
15  0 0 0 0
15.1  1 1 0 1
15.8  1 0 0 0

; STORE 01 6
17  0 0 0 0
17.1  1 1 0 1
17.8  1 0 0 0

; STORE 02 5
23  0 0 0 0
23.1  1 1 0 1
23.8  1 0 0 0

; STORE 03 6
24  0 0 0 0
24.1  1 1 0 1
24.8  1 0 0 0

; STORE 04 7
26  0 0 0 0
26.1  1 1 0 1
26.8  1 0 0 0

; LOAD 0 00
27  0 0 0 0
27.1  1 0 0 1
27.6  1 0 1 0
27.9  1 0 0 0

; LOAD 0 01
28  0 0 0 0
28.1  1 0 0 1
28.6  1 0 1 0
28.9  1 0 0 0

; LOAD 0 02
29  0 0 0 0
29.1  1 0 0 1
29.6  1 0 1 0
29.9  1 0 0 0

; LOAD 0 03
30  0 0 0 0
30.1  1 0 0 1
30.6  1 0 1 0
30.9  1 0 0 0

; LOAD 0 04
31  0 0 0 0
31.1  1 0 0 1
31.6  1 0 1 0
31.9  1 0 0 0
```

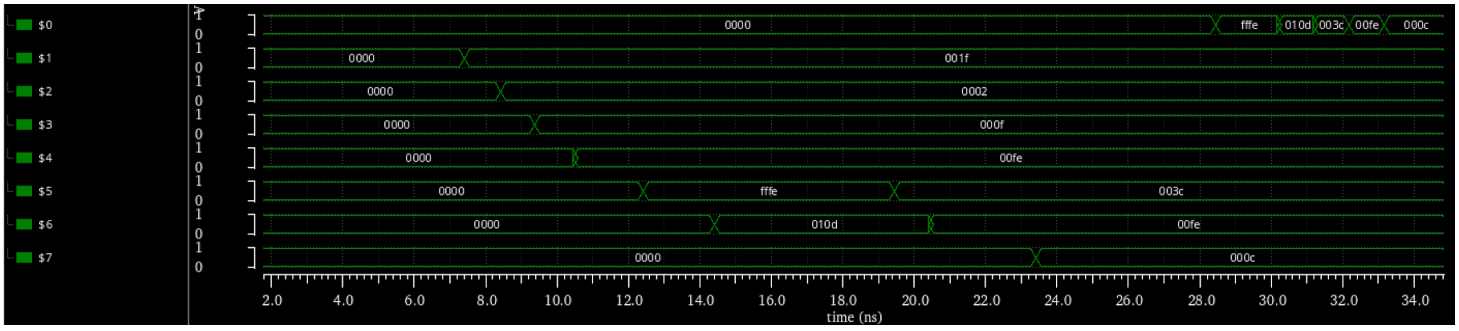
- Back-end generated golden results: **golden_results.txt**

golden_results.txt

```
Register File - Final Content:
$0: 000C
$1: 001F
$2: 0002
$3: 000F
$4: 00FE
$5: 003C
$6: 00FE
$7: 000C

SRAM - Final Content:
00H: FFFE
01H: 010D
02H: 003C
03H: 00FE
04H: 000C
0AH: 001F
0BH: 0002
10H: 000F
11H: 00FE
```

- Final content in Register File after Cadence simulation on hardware top-level schematic

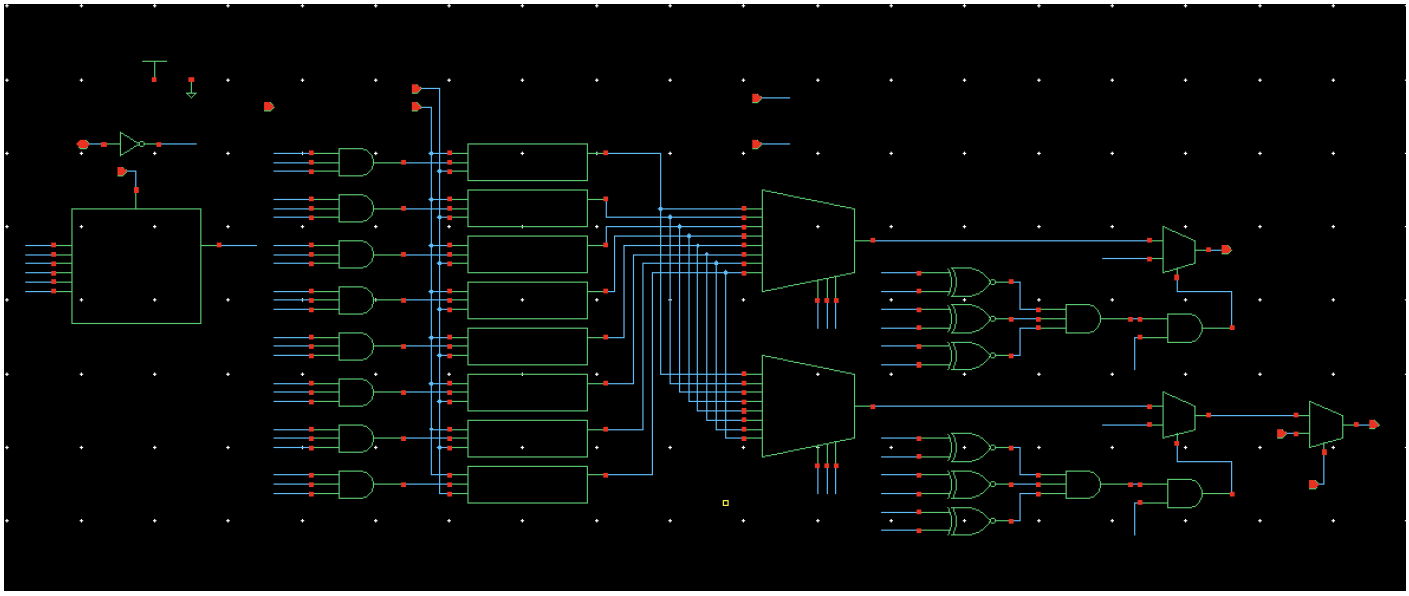


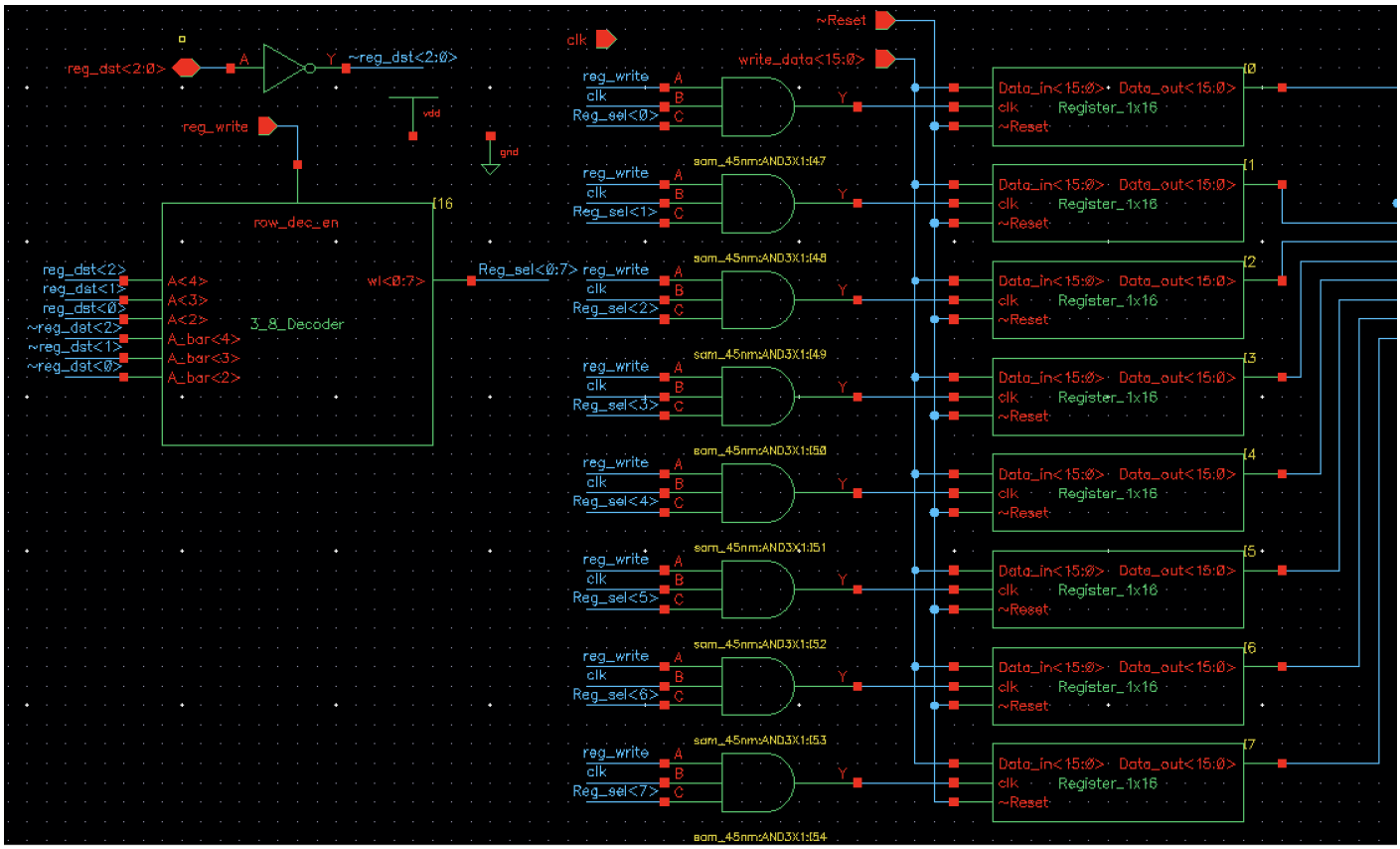
2. ID

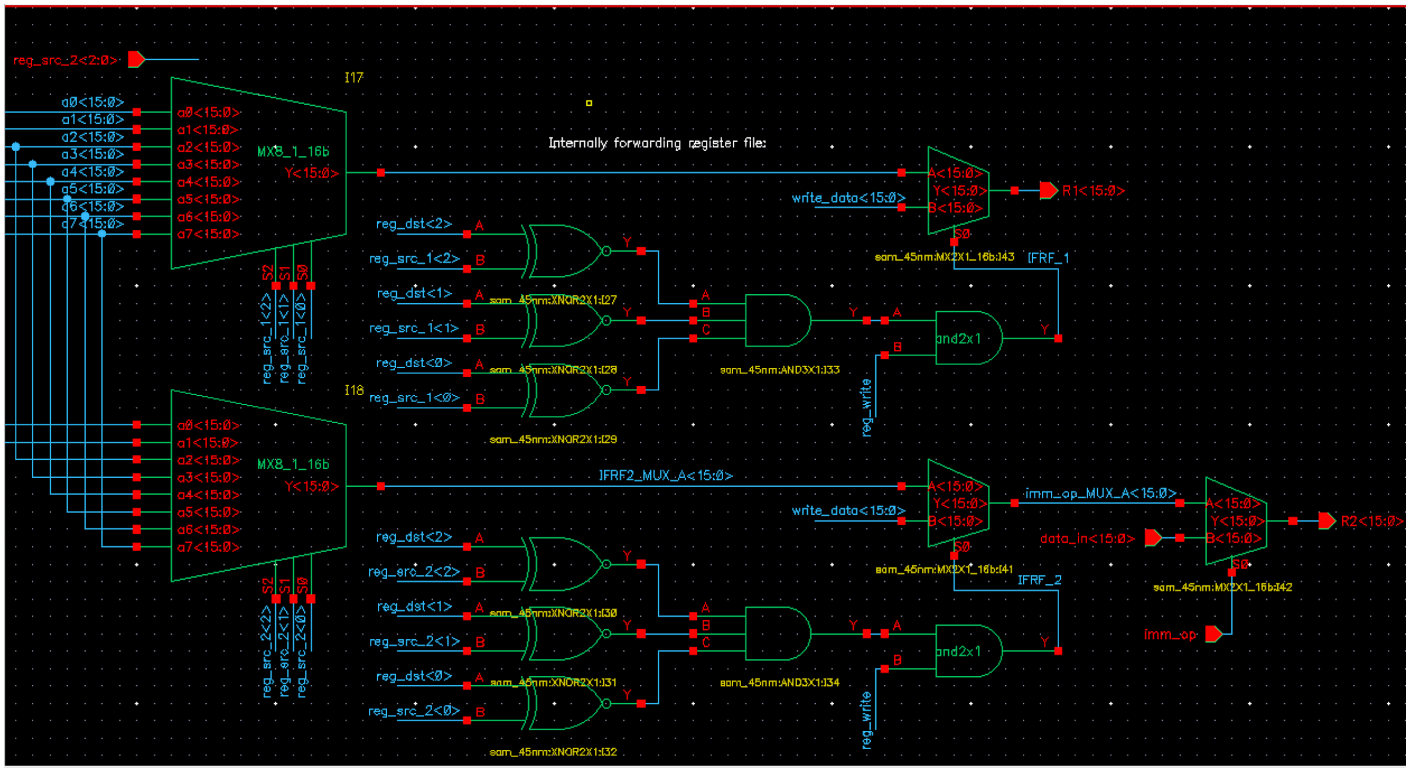
- RF

  According to the block diagram provided in the project description, the ID stage is divided into two parts. The front end python, as mentioned above, handles the decode portion while the register file is implemented in hardware. The following 3 schematics highlight the register file implementation. The register file memory is implemented using DFFs. There are 8 16 bit registers ($0-$7). The register file supports reading two registers simultaneously by multiplexing the register outputs and selecting a register with the respective source destination. Additionally, the register file supports an internally forwarding register file which is an optimization to save a NOP in phase 2. While

this is implemented in HW, our current python script does not support its verification. This functions by comparing the destination register with the respective source register. If these match and the reg_write signal is asserted, the RF will write this input data to the respective register while also outputting it to the respective RF output line. By default, there also exists a multiplexer on the register source 2 output to select between the register source 2 data or the immediate value for I-type instructions.

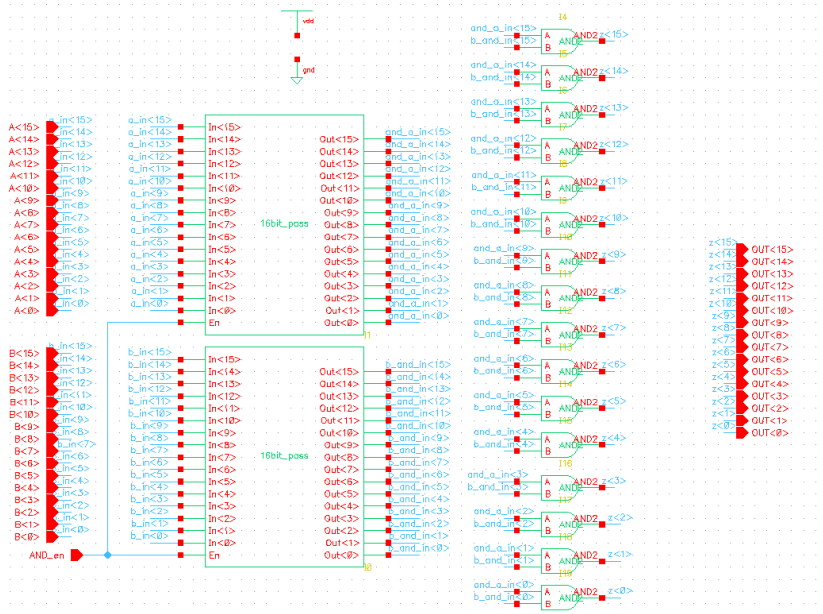




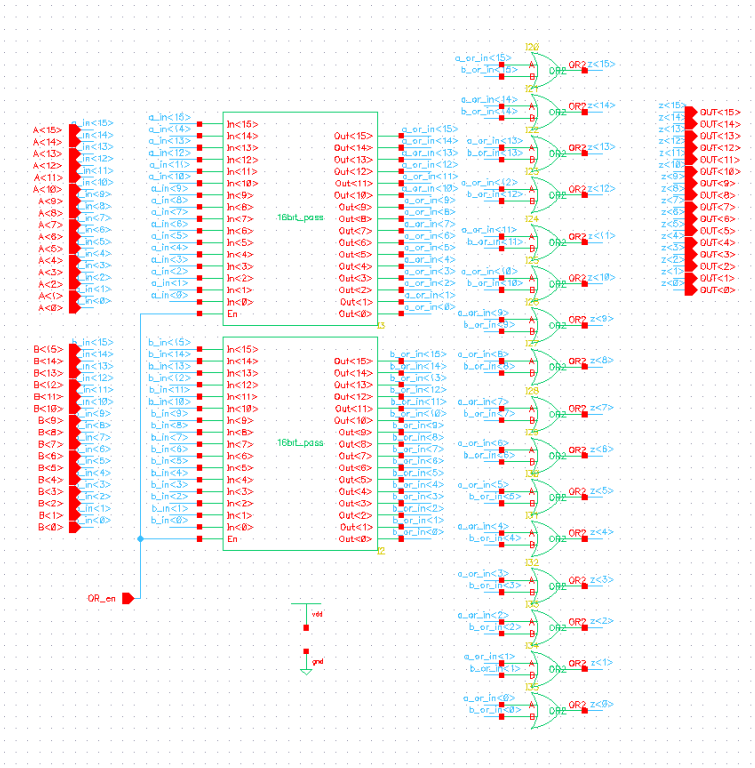


3. EX

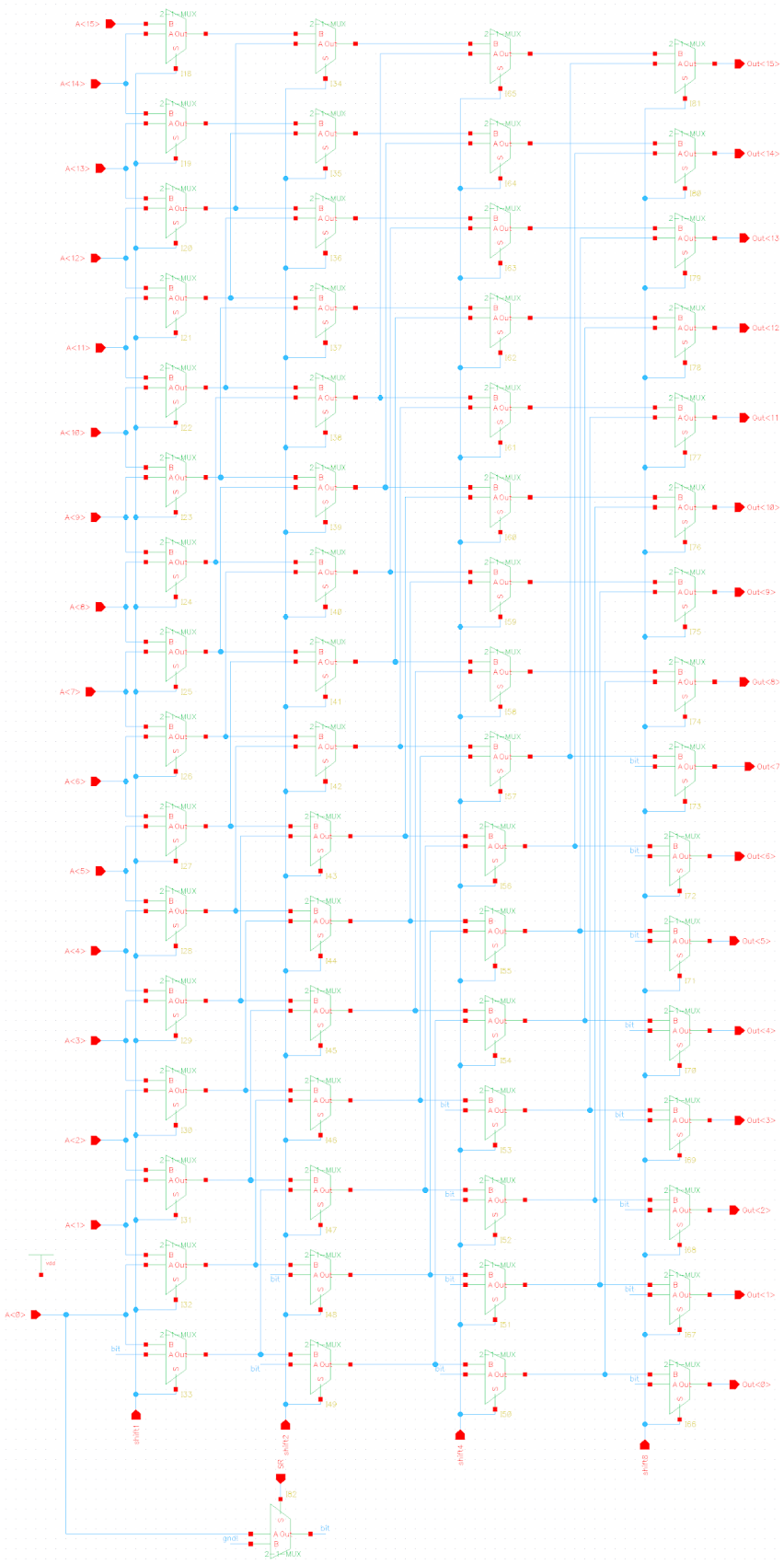
- Bitwise AND

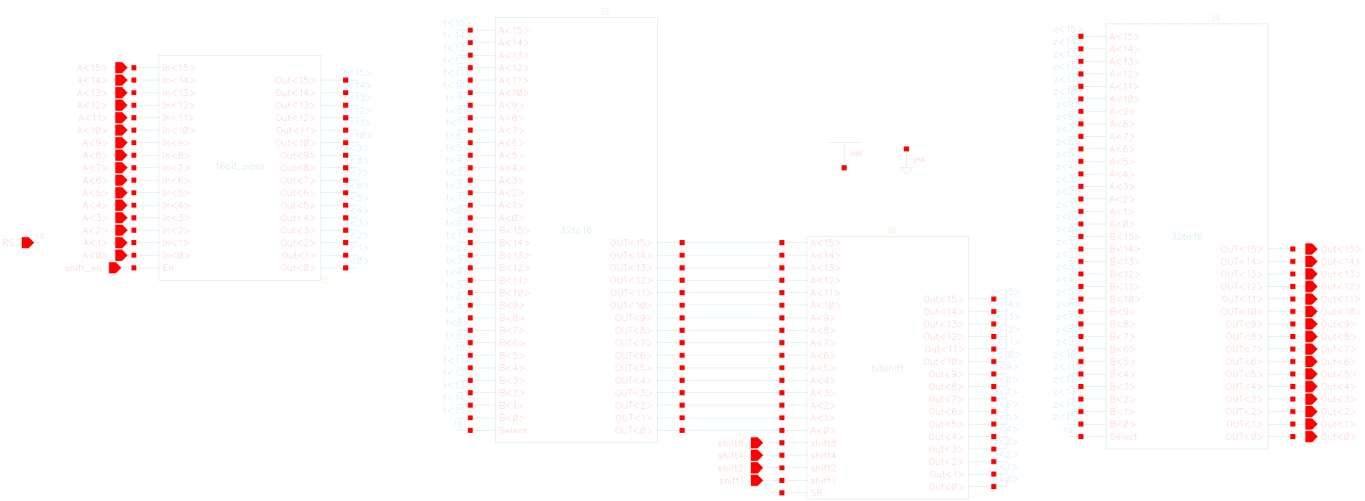


- Bitwise OR



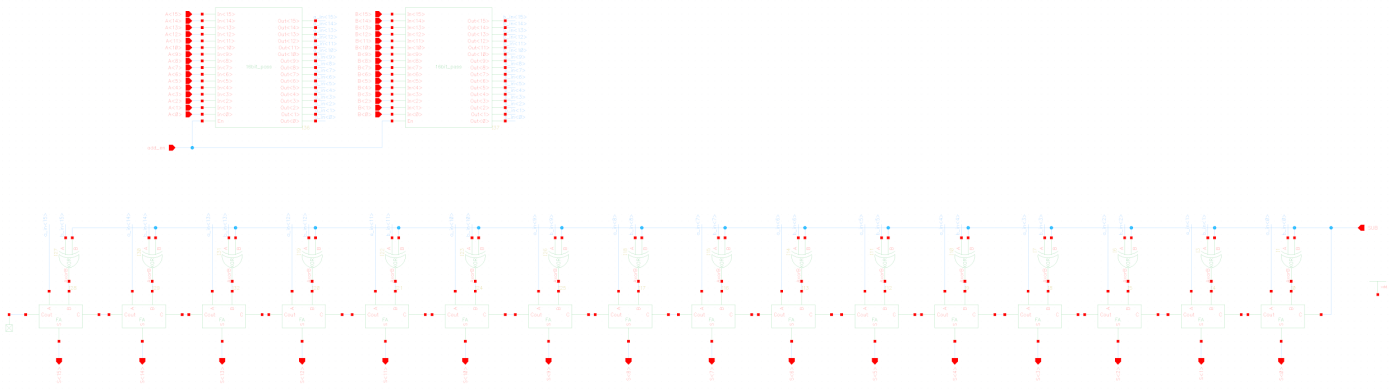
- SFL/SFR


A<15>
A<14>
A<13>
A<12>
A<11>
A<10>
A<9>
A<8>
A<7>
A<6>
A<5>
A<4>
A<3>
A<2>
A<1>
A<0>
Out<15>
Out<14>
Out<13>
Out<12>
Out<11>
Out<10>
Out<9>
Out<8>
Out<7>
Out<6>
Out<5>
Out<4>
Out<3>
Out<2>
Out<1>
Out<0>
shift1
shift2
shift3
shift4
SR
gnd!
vdd
bit
MUX



- ADD/MIN

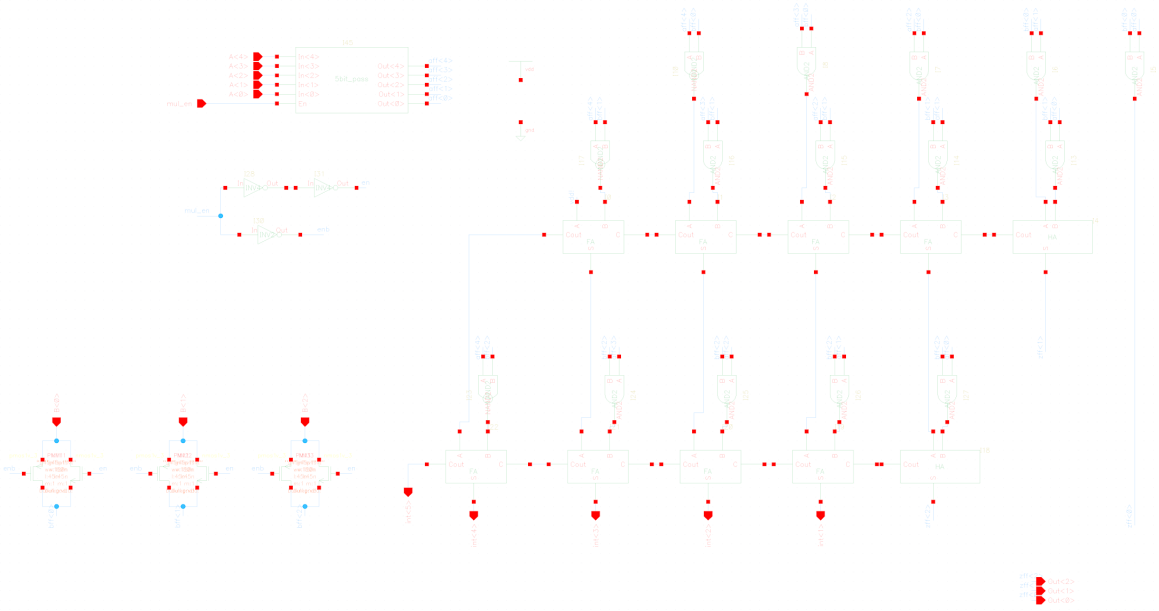
Depending on the value of the input carry, this unit can function as an adder or a subtractor. Subtractor is used to find the minimum value of two inputs. Then, the MSB is used to select that minimum value. Since the multiplier is the bottleneck for the clock frequency, using a Carry Propagate Adder/Subtractor topology does not cause issues with the CPU speed.



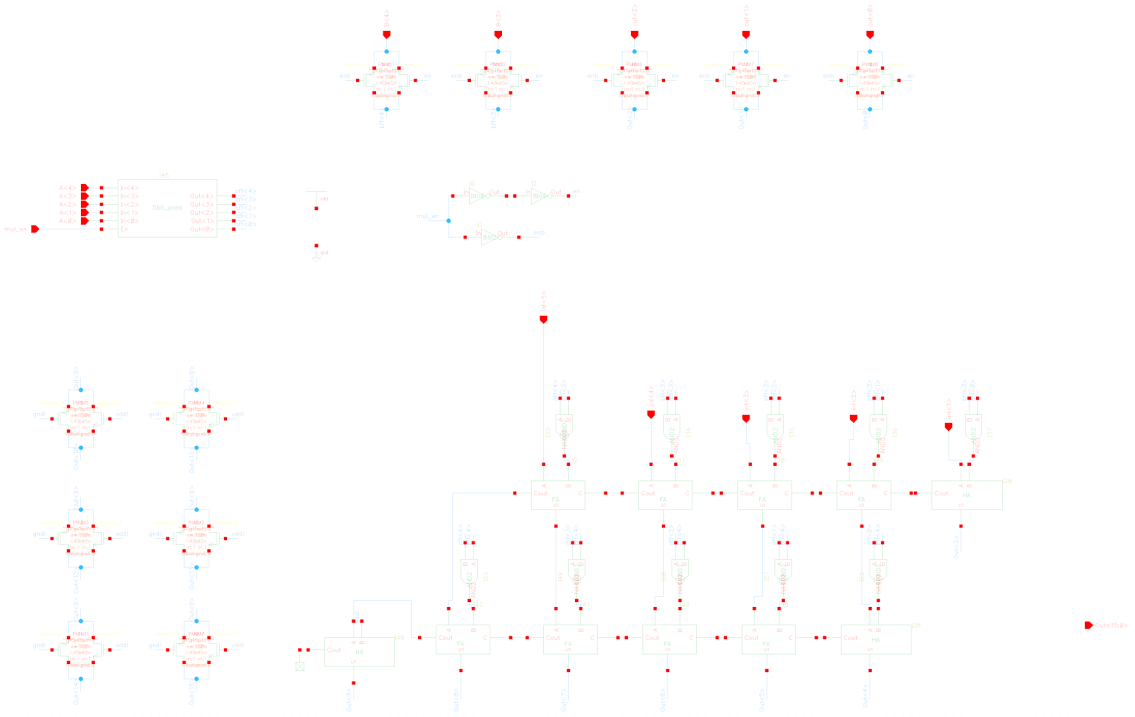
- MUL

We split the multiplier into two blocks - one of them resides in EX and another in MEM stage, respectively. This approach allows us to boost the clock frequency significantly because the propagation delay of the single multiplier limits the maximum clock frequency that ensures a proper operation of the CPU. Since 5x5 multiplication gives 10 bits at the output, our configuration takes care of remaining <15:10> bits by replicating the MSB of the original multiplication product.

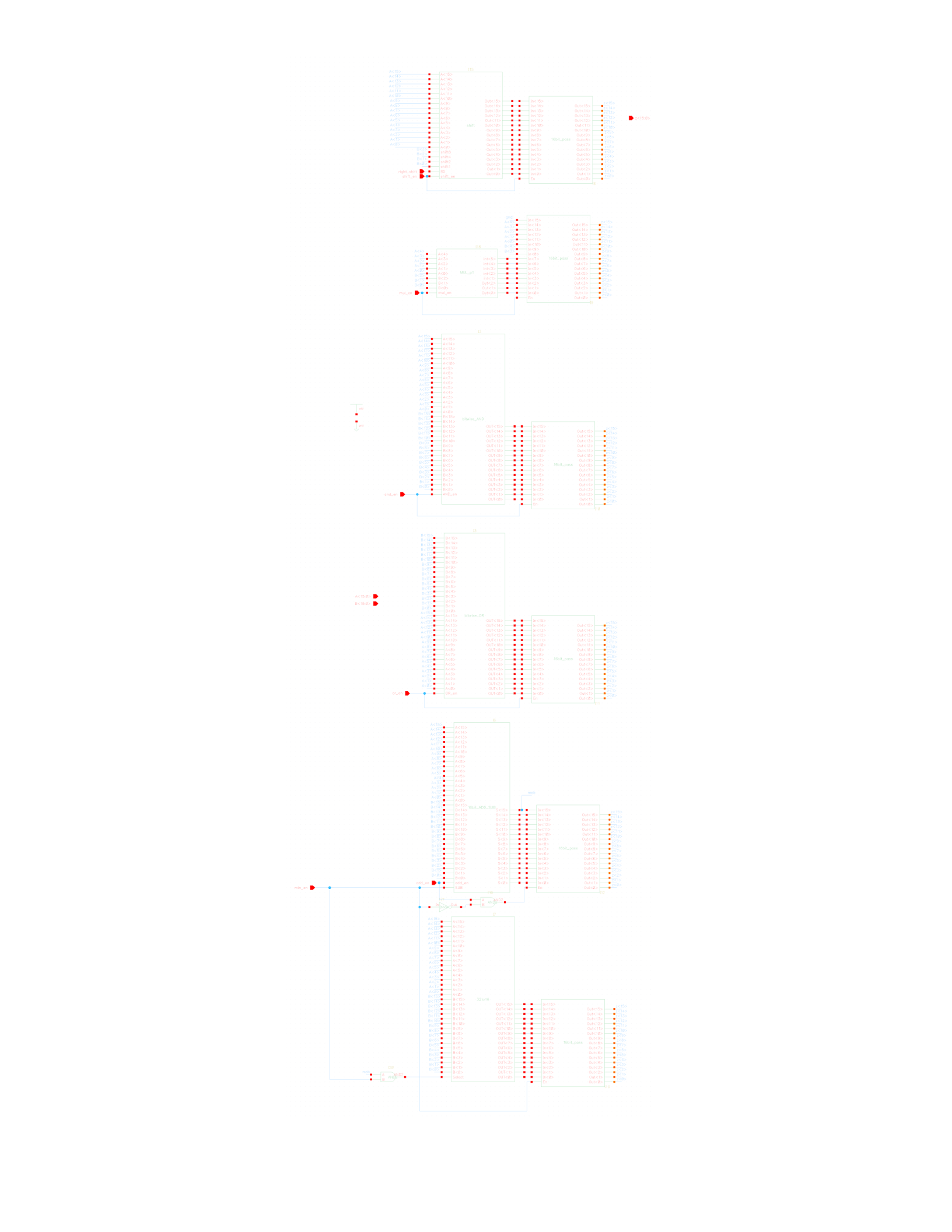
Part 1



Part 2



- Whole EX block

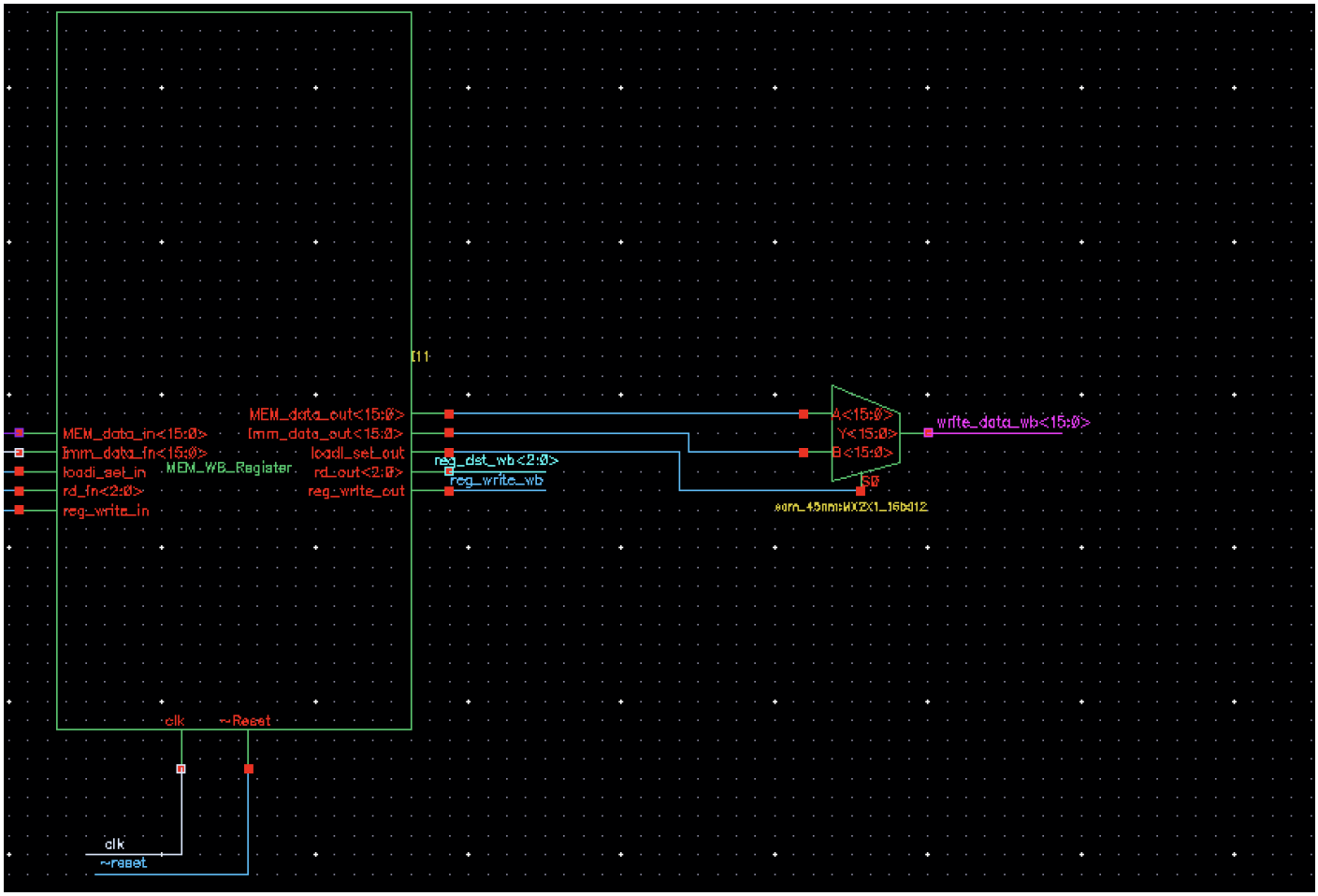


4. MEM

An SRAM was directly adopted from the Lab2 assignment.

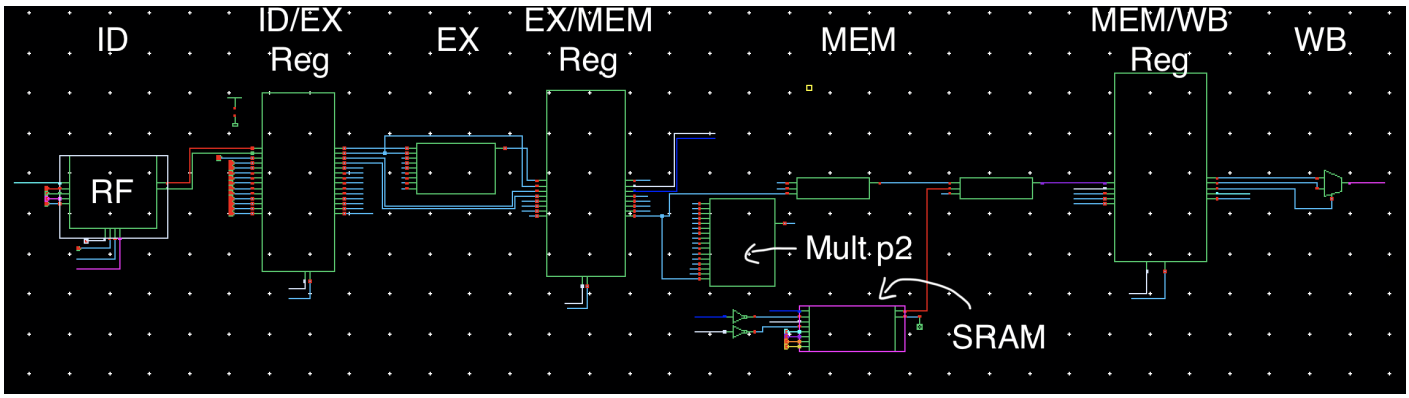
5. WB

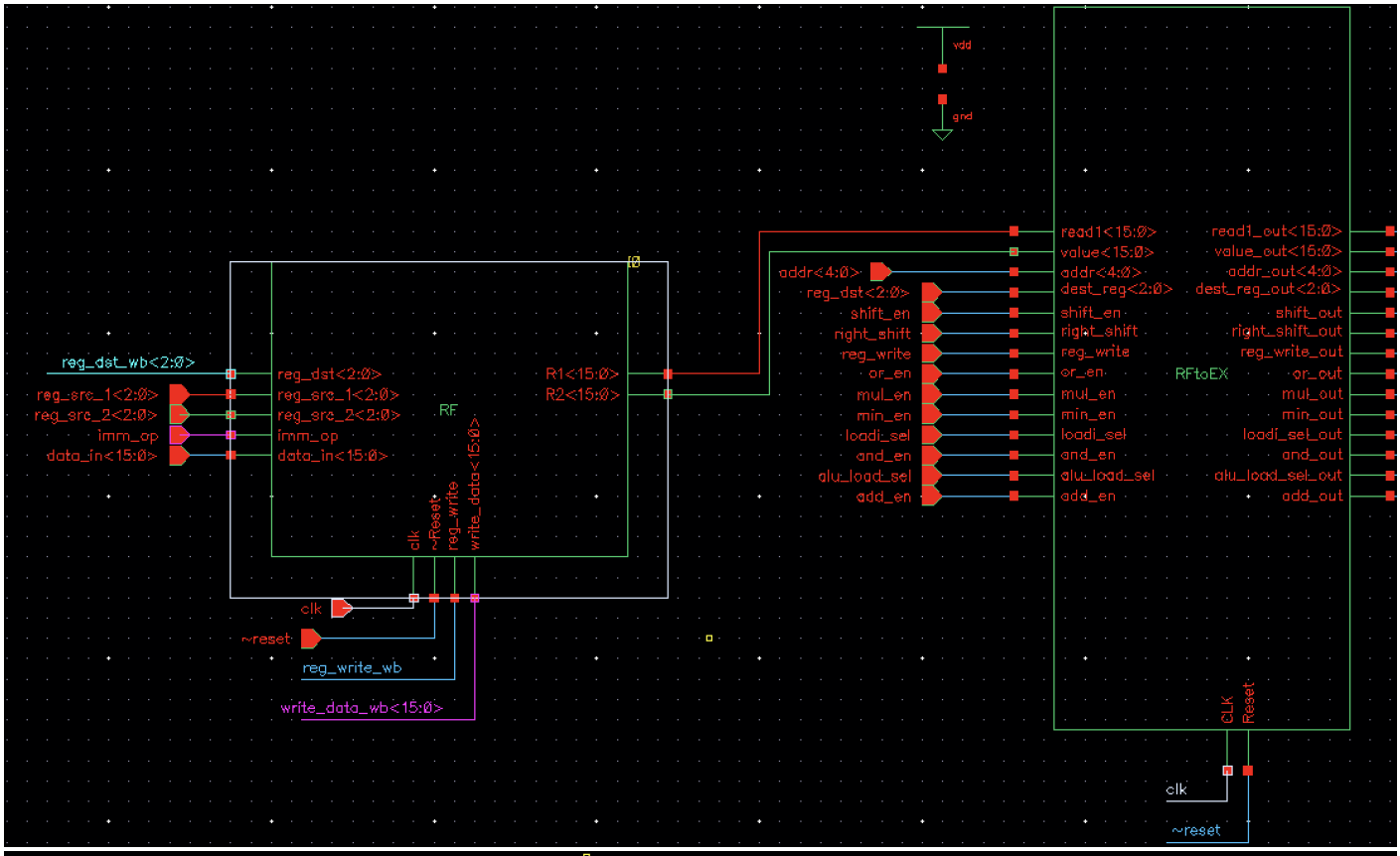
The write back stage is a simple implementation. It selects between the 16bit data output of the MEM stage (either ALU or SRAM output) or an immediate 16bit value and passes this value to the data input of the register file to be written at reg_dst_wb if reg_write_wb is high.

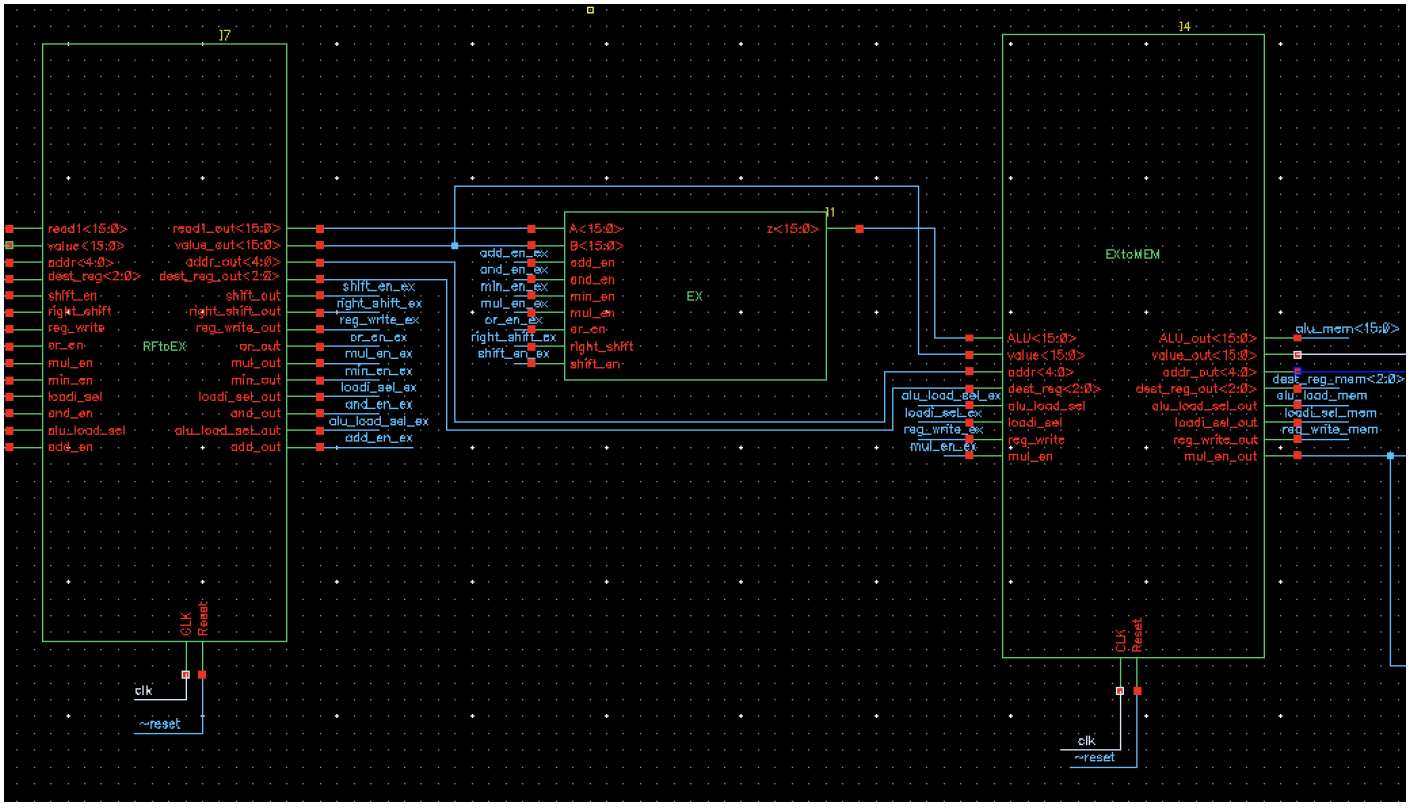


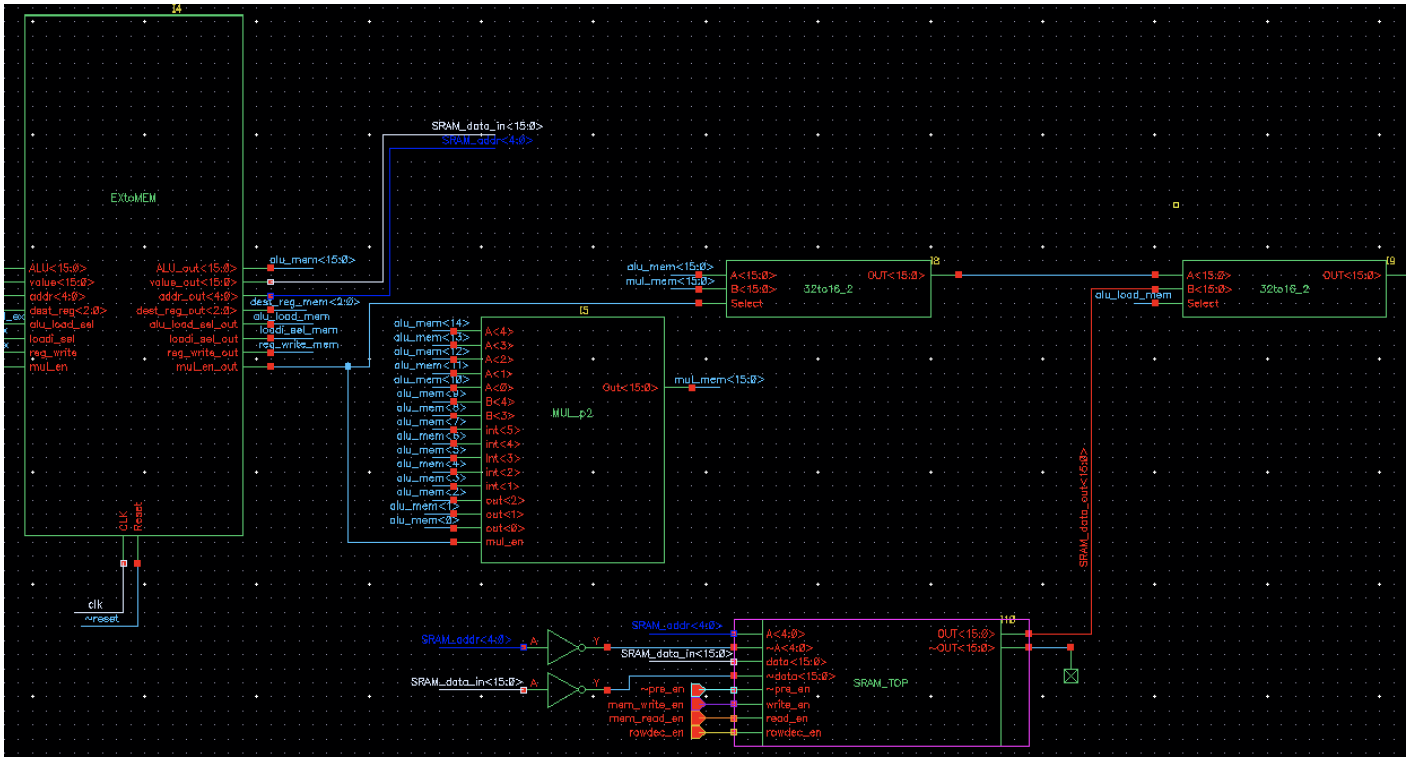
6. System Integration

The following screenshots show the high level system integration block with each major module as well as a zoomed in detailed view of each pipeline stage.


ID
ID/EX Reg
EX
EX/MEM Reg
MEM
MEM/WB Reg
WB
RF
Mult p2
SRAM


vdd
gnd
I0
reg_dst_wb<2:0>
reg_src_1<2:0>
reg_src_2<2:0>
imm_op
data_in<15:0>
reg_dst<2:0>
reg_src_1<2:0>
reg_src_2<2:0>
imm_op
data_in<15:0>
RF
R1<15:0>
R2<15:0>
clk
~Reset
reg_write
write_data<15:0>
clk
~reset
reg_write_wb
write_data_wb<15:0>
addr<4:0>
reg_dst<2:0>
shift_en
right_shift
reg_write
or_en
mul_en
min_en
loadi_sel
and_en
alu_load_sel
add_en
read1<15:0>
value<15:0>
addr<4:0>
dest_reg<2:0>
shift_en
right_shift
reg_write
or_en
mul_en
min_en
loadi_sel
and_en
alu_load_sel
add_en
RFtoEX
read1_out<15:0>
value_out<15:0>
addr_out<4:0>
dest_reg_out<2:0>
shift_out
right_shift_out
reg_write_out
or_out
mul_out
min_out
loadi_sel_out
and_out
alu_load_sel_out
add_out
CLK
Reset
clk
~reset


I7
read1<15:0>
value<15:0>
addr<4:0>
dest_reg<2:0>
shift_en
right_shift
reg_write
or_en
mul_en
min_en
loadi_sel
and_en
alu_load_sel
add_en
RFtoEX
read1_out<15:0>
value_out<15:0>
addr_out<4:0>
dest_reg_out<2:0>
shift_out
right_shift_out
reg_write_out
or_out
mul_out
min_out
loadi_sel_out
and_out
alu_load_sel_out
add_out
shift_en_ex
right_shift_ex
reg_write_ex
or_en_ex
mul_en_ex
min_en_ex
loadi_sel_ex
and_en_ex
alu_load_sel_ex
add_en_ex
CLK
Reset
clk
~reset
I1
A<15:0>
B<15:0>
add_en
and_en
min_en
mul_en
or_en
right_shift
shift_en
EX
z<15:0>
add_en_ex
and_en_ex
min_en_ex
mul_en_ex
or_en_ex
right_shift_ex
shift_en_ex
alu_load_sel_ex
loadi_sel_ex
reg_write_ex
mul_en_ex
I4
EXtoMEM
ALU<15:0>
value<15:0>
addr<4:0>
dest_reg<2:0>
alu_load_sel
loadi_sel
reg_write
mul_en
ALU_out<15:0>
value_out<15:0>
addr_out<4:0>
dest_reg_out<2:0>
alu_load_sel_out
loadi_sel_out
reg_write_out
mul_en_out
alu_mem<15:0>
dest_reg_mem<2:0>
alu_load_mem
loadi_sel_mem
reg_write_mem
CLK
Reset
clk
~reset


I4
EXtoMEM
ALU<15:0>
value<15:0>
addr<4:0>
dest_reg<2:0>
alu_load_sel
loadi_sel
reg_write
mul_en
ALU_out<15:0>
value_out<15:0>
addr_out<4:0>
dest_reg_out<2:0>
alu_load_sel_out
loadi_sel_out
reg_write_out
mul_en_out
clk
reset
clk
~reset
SRAM_data_in<15:0>
SRAM_addr<4:0>
alu_mem<15:0>
dest_reg_mem<2:0>
alu_load_mem
loadi_sel_mem
reg_write_mem
I5
MUL_p2
mul_mem<15:0>
I8
32to16_2
I9
SRAM_data_out<15:0>
I10
SRAM_TOP
SRAM_addr<4:0>
SRAM_data_in<15:0>
~pre_en
mem_write_en
mem_read_en
rowdec_en
write_en
read_en
OUT<15:0>
~OUT<15:0>
I11
MEM_data_in<15:0>
SRAM_data_in<15:0>
loadi_sel_mem
dest_reg_mem<2:0>
reg_write_mem
MEM_WB_Register
MEM_data_out<15:0>
mm_data_out<15:0>
loadi_sel_out
rd_out<2:0>
reg_write_out
reg_dst_wb<2:0>
reg_write_wb
write_data_wb<15:0>
clk
~Reset
clk
~reset