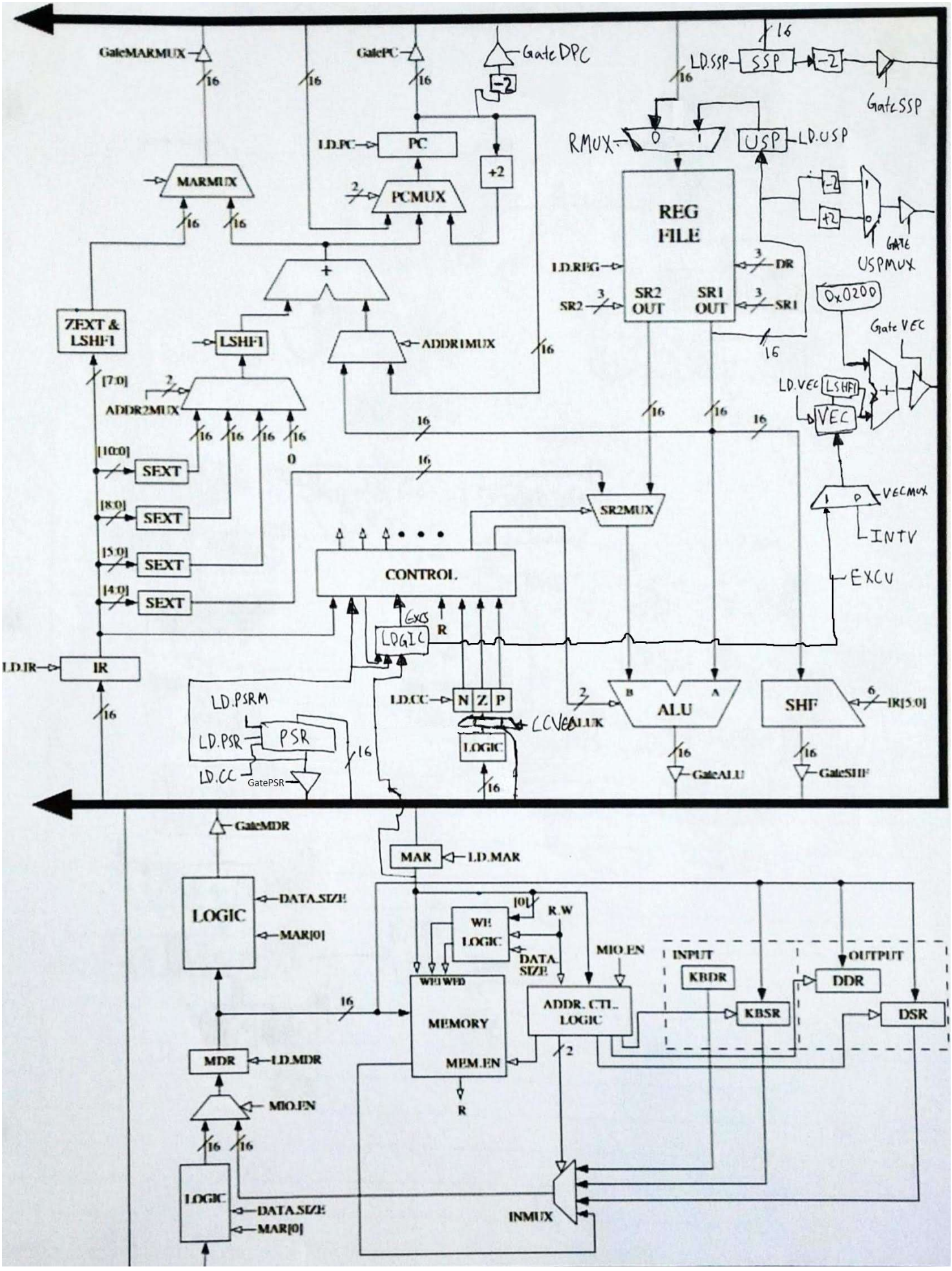
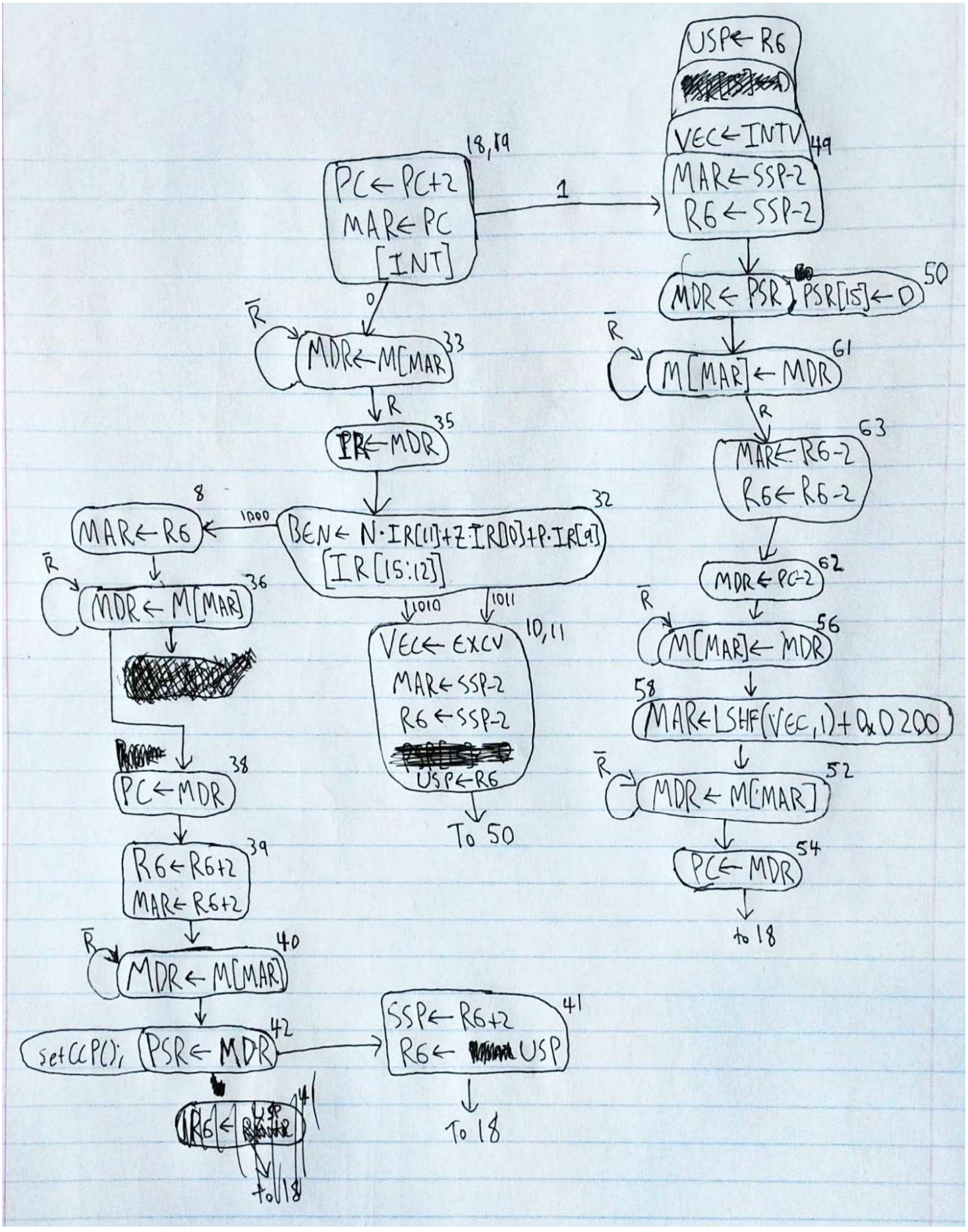
Suhas Raja

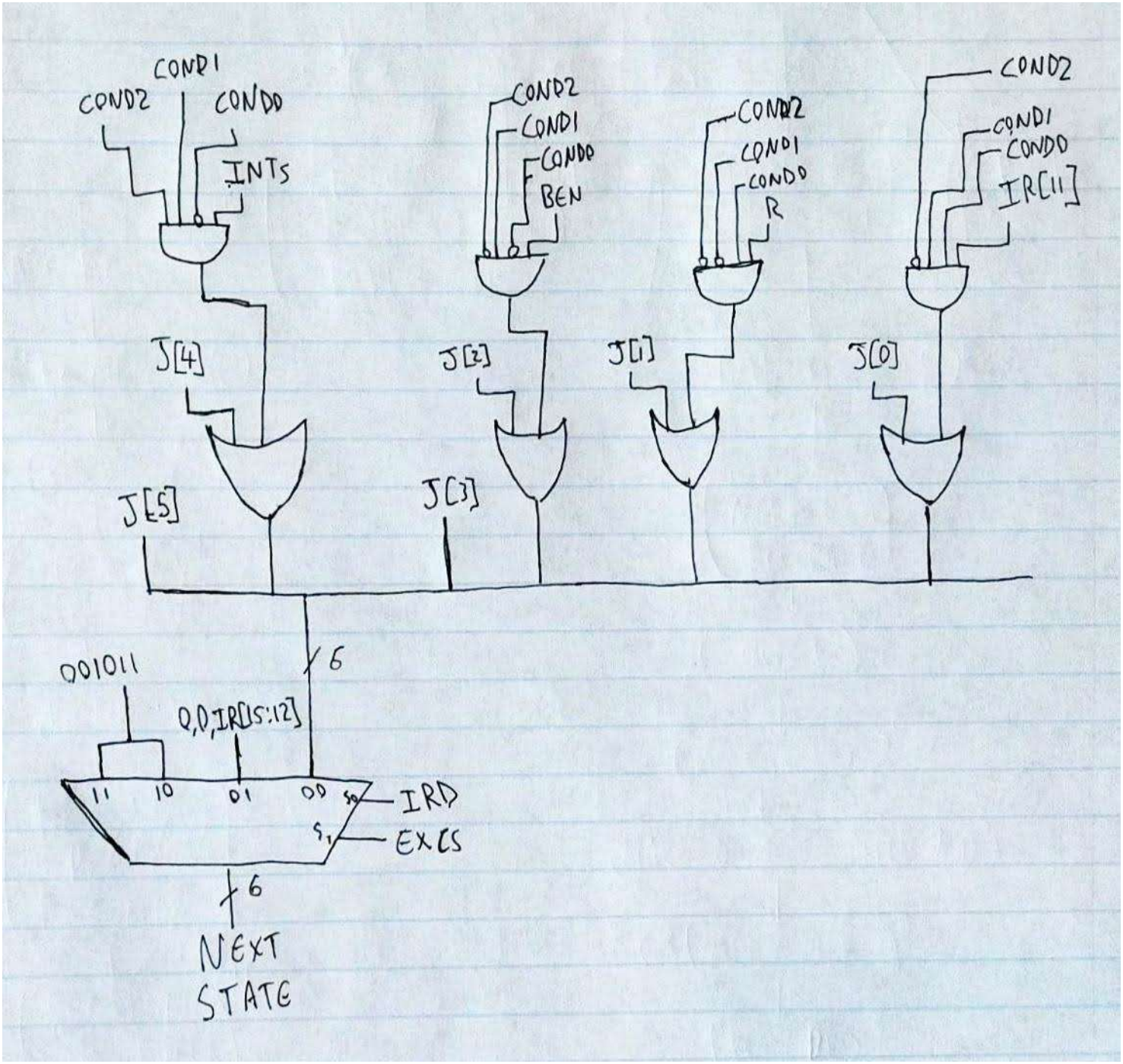
EE460N Lab 4 Documentation

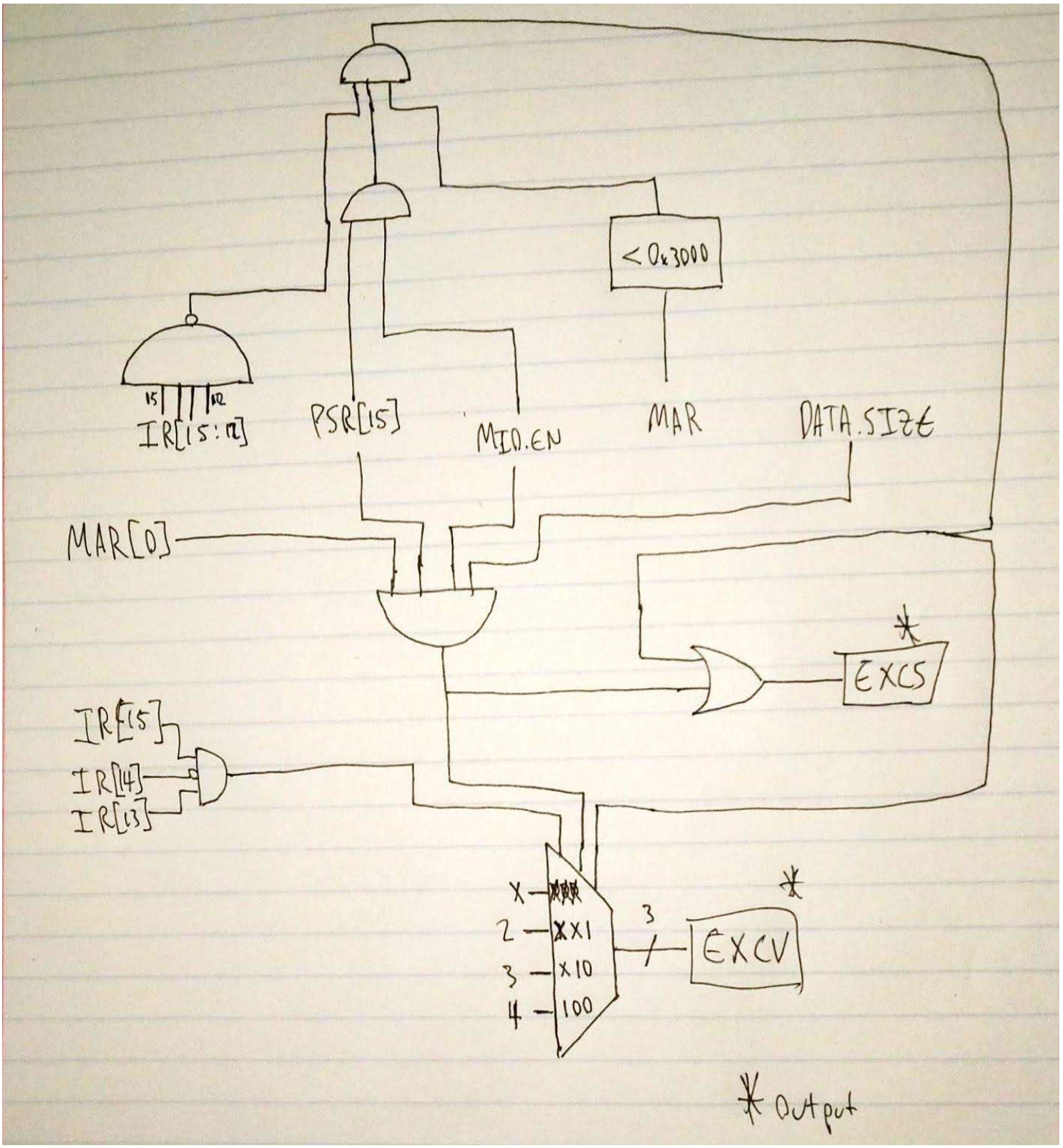
- FSM Modifications
  - Interrupts
    - Interrupts are detected when the INTS signal goes high. Only state 18 is sensitive to the INTS signal. When the simulator detects the interrupt in state 18, it will branch to state 49 instead of 33 (per the microsequencer). State 49 saves the interrupt vector to the VEC register, saves the User Stack Pointer, and loads the destination address on the stack to R6 and the MAR. The next state copies the PSR to the MDR in preparation for pushing it on the stack, and clears bit 15 of the PSR to indicate Supervisor mode. After storing the PSR to the stack, R6 and the MAR are then decremented by 2, the MDR is loaded with PC-2 (the -2 ensures the current instruction will be executed post-interrupt), and the store on the stack occurs. Next, the hardware loads 0x0200+2*VEC to calculate the pointer to the ISR, loads this value to the MAR, then performs a memory access to bring the ISR pointer into the PC. The machine is now in state 18 and pointing at the ISR, so the interrupt has now been successfully completed.
  - RTI
    - When RTI is executed, the system moves to state 8 for the execution phase. The RTI states pop the PC and PSR off the stack, in that order, save R6 to the SSP register, and restores the USP into R6. While the PSR is being popped off the stack, it passes through the bus into the PSR register. While it is on the bus, the condition code mux (controlled by CCMUX) makes the NZP registers peek at the PSR while on the bus to restore the system state accordingly, using direct wire connections.
  - Exceptions
    - Exceptions begin in state 10 or 11 (identical). An unknown opcode exception may be naturally initiated by the microsequencer setting next_state to IR[15:12], since 1010 and 1011 are both unknown opcodes. Otherwise, the unaligned access and protection exceptions are generated by logic hardware which sets the EXCS high. This logic reads from the PSR, control store, IR, and MAR, to see if either a.) a word sized load/store is occurring in this state and the MAR[0]==1 or b.) PSR[15]==0, a memory access is about to occur (as determined by MIO.EN), and MAR<0x3000. This logic is shown on page 6, and generates the EXCS signal as well as the EXCV vector that is passed into the VEC register. Once the EXCS signal goes high, the microsequencer mux ignores the J bits and IR and instead sets next_state to 11, where the exception begins. This state sets VEC as the relevant exception vector, prepares R6 and the MAR as the decremented Supervisor Stack Pointer, saves the User Stack Pointer, then branches to the interrupt states that service the pushing of VEC and PC-2 on the stack in addition to moving system & PC to the service routine.
- Microsequencer
  - The INTS signal may set up bit 4 of next_state if COND2 (a new control signal) is high. This allows the FSM to begin servicing an interrupt once it is in state 18 and INTS is high.
  - I passed the EXCS signal from the datapath into the mux at the base of the microsequencer. This means that the next time next_state is computed after EXCS goes high, next_state is hardset at state 11, the first state for exceptions.
- Hardware Added
  - I added the SSP and USP registers to the datapath. These track the most recent value of the Supervisor Stack Pointer and User Stack Pointer, respectively, from the last time the system was in the according mode (Supervisor or User mode). This means that I needed to add signals to write to them and gate them onto the bus (LD.SSP/LD.USP, GateSSP/GateUSP). The only time we read from the SSP in the FSM is when we are reading the value SSP-2, so there is a block to decrement by 2 before being gated to the bus. The USP does not connect to the bus, since it only ever reads and writes to R6. I connected the USP load directly to the SR1 output, which may point directly at

R6 based on the SR1MUX1 signal. Similarly, I added a mux at the load at the register to determine whether the register file reads a value in from the bus or the USP. This mux admittedly increases the propogation delay in register writes from the bus.
    - I also added a path that bypasses the ALU to dump R6 directly on the bus. This is useful since it doesn't dump the original value of R6; rather, it dumps R6+2 or R6-2, depending on the value of SPMUX. It only gets dumped if the GateSPS signal is high. This path is useful so we can increment/decrement the current SP with dedicated hardware in the datapath in a single cycle. Similarly, it would make it easier to create future functionality for PUSH/POP instructions in LC3.
    - I added a similar path from the current PC through a block that decrements by 2, and dumps PC-2 to the bus based on the GateDPC signal. This allows the interrupt/exception states to reset the PC to the problematic instruction with ease, since the PC doesn't have any elegant connection through arithmetic components in the original datapath.
    - I added a PSR register to the datapath to track the value of the register as the system undergoes context switches. This required a LD.PSR signal to write a value to the PSR from the bus. Since we often need to set the most significant bit of the PSR to 0 without changing the bits, I used the LD.PSRM bit to clear bit 15 of the PSR without using the bus. The PSR may also be gated onto the bus using the GatePSR signal
    - I added a VEC register that stores the most recent vector that would be necessary for servicing an Interrupt or Exception. This required a LD.VEC signal to write to this register and a mux that writes the value from either INTV or EXCV depending on the VECMUX signal. The value of this vector is leftshifted by 1 and added to 0x0200 to find the address of the necessary service routine, and is only gated onto the bus if GateVEC is 1. This would occur if we wanted to write that address to the MAR.
    - I also added combinational logic hardware that reads from the current control signals, MAR value, and PSR value, to generate the EXCS signal that is passed into the FSM/Control box and the EXCV vector. This is used to generate exceptions.
    - I added a mux before the condition codes, which changes whether the condition codes are set by logic reading from the bus, or the 3 least significant bits passed directly into the register.
- All Control Signals Added
    - COND2- Required for microsequencer integration of INTS signal.
    - LD.SSP- Indicates a write to SSP register.
    - LD.USP- Indicates a write to USP register.
    - LD.PSR- Indicates a write to PSR register.
    - LD.PSRM- Clears PSR[15].
    - LD.VEC- Indicates a write to VEC register.
    - GateSSP- Indicates whether SSP-2 is dumped to bus.
    - GateSPS- Indicates whether current stack pointer, either incremented or decremented by 2, is dumped to bus.
    - GateDPC- Indicates whether decremented PC (PC-2) is dumped to bus.
    - GatePSR- Indicates whether PSR register is dumped to bus.
    - GateVEC- Indicates whether VEC register is dumped to bus.
    - DRMUX1- If it is high, then DR must be 6.
    - SR1MUX1- If high, then SR must be 6.
    - RMUX- If high, then register file loads from USP. Otherwise, register file loads from bus.
    - USPMUX- Determines whether current SP is incremented or decremented before being dumped to the bus.
    - VECMUX- Determines whether EXCV or INTV is written to VEC register.
    - CCMUX- Determines whether logic sets the condition codes, or wires to BUS[2:0].
- FSM Shorthand
    - setcc2()- denotes that the condition codes are set based on the PSR value being passed through the bus.
    - States for servicing normal instructions are not shown for simplicity.
    - [INT] indicates the next state is dependent on the INTS signal.


GateMARMUX
GatePC
GateDPC
LD.SSP
SSP
-2
GateSSP
RMUX
USP
LD.USP
LD.PC
PC
+2
MARMUX
PCMUX
REG FILE
LD.REG
DR
SR2
SR2 OUT
SR1 OUT
SR1
GATE USPMUX
0x0200
Gate VEC
LD.VEC
LSHF1
VEC
ZEXT & LSHF1
LSHF1
ADDR1MUX
ADDR2MUX
SEXT
SR2MUX
VECMUX
INTV
EXCV
CONTROL
LD.IR
IR
LDGIC
LD.PSRM
LD.PSR
PSR
LD.CC
GatePSR
LD.CC
N Z P
LOGIC
ALU
SHF
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
WE LOGIC
R.W
MIO.EN
INPUT
OUTPUT
KBDR
DDR
KBSR
DSR
MEMORY
MEM.EN
ADDR. CTL. LOGIC
MDR
LD.MDR
MIO.EN
INMUX


18,19
PC ← PC+2
MAR ← PC
[INT]
1
0
R̄
MDR ← M[MAR]
33
R
IR ← MDR
35
32
BEN ← N·IR[11]+Z·IR[10]+P·IR[9]
[IR[15:12]]
1000
8
MAR ← R6
R̄
MDR ← M[MAR]
36
38
PC ← MDR
39
R6 ← R6+2
MAR ← R6+2
R̄
MDR ← M[MAR]
40
setCC();
PSR ← MDR
42
41
SSP ← R6+2
R6 ← USP
To 18
1010
1011
10,11
VEC ← EXCV
MAR ← SSP-2
R6 ← SSP-2
USP ← R6
To 50
USP ← R6
VEC ← INTV
49
MAR ← SSP-2
R6 ← SSP-2
MDR ← PSR
PSR[15] ← 0
50
R̄
M[MAR] ← MDR
61
R
MAR ← R6-2
R6 ← R6-2
63
MDR ← PC-2
62
R̄
M[MAR] ← MDR
56
58
MAR ← LSHF(VEC,1)+0x0200
R̄
MDR ← M[MAR]
52
PC ← MDR
54
to 18


COND1
COND2
COND0
INTS
J[4]
COND2
COND1
COND0
BEN
J[2]
COND2
COND1
COND0
R
J[1]
COND2
COND1
COND0
IR[11]
J[0]
J[5]
J[3]
6
001011
0,0,IR[5:12]
11
10
01
00
IRD
EXCS
6
NEXT
STATE


< 0x3000
15
12
IR[15:12]
PSR[15]
MIO.EN
MAR
DATA.SIZE
MAR[0]
EXCS
*
IR[15]
IR[14]
IR[13]
X
2
3
4
XXX
XX1
X10
100
3
EXCV
*
* Output