Name: Aria Pahlavan

EID: ap44342

1. Changes you made to the state diagram. Include a picture similar to the state machine that shows the new states you added.

First, I added all the states required to properly execute the RTI instruction. To do so, I have added 8 new states (including state 8 which is for RTI instruction). In states 8 and 36, I am popping the first item on the state, which is the PC, by loading R6 (i.e. the stack pointer) into MAR, so I can load the contents of that memory location to MDR. Then in state 38 I am loading the PC with the PC of the interrupted program. States 39, 62, and 63, acheive the same goal by incrementing the stack pointer and loading MDR with the contents of that memory location, which is the original PSR before interrupt occurred. In states 34 and 59, I increment the stack pointer once more and then save it in the Saved Supervisor Stack Pointer register and reload the stack pointer with the Saved User Stack Pointer Register, which was saved before initiating the interrupt or exception.

Second, I have added all the states required to properly execute interrupts or exceptions. State 41 is only reached from states 18, 19 if and only if there is an interrupt and the current privilege is User Mode. That is because interrupts only checked before executing the next instruction. State 41 only loads the Vector register with the interrupt vector index (it becomes a full address in another state). State 49 can be reached in case of an interrupt or an exception to load MDR with current PSR and set privilege mode to to 0 (Supervisor Mode). State 56 simply saves the current stack pointer for user mode and loads the supervisor stack pointer from its register. States 58 and 60 perform pushing of the old PSR on the supervisor stack by decrementing the stack pointer and the loading the contents of the memory on the stack. States 61, 47, and 44 perform the same operation except this time we're pushing the decremented PC on the stack, so we can execute the same instruction after the handling is over. State 45 adds the Vector register to the base of vector table to load MDR with the address of the proper vector routine from the memory in state 52 (or

from the interrupt vector table). Finally, in state 54, PC is loaded with the correct address of service routine.

Last, I added the states for checking the protection, unaligned access, and illegal opcode exception. I updated states 10 and 11 so that the vector register is loaded with the index for illegal opcode exception (x04) and made it point to state 49 so the exception is executed properly. I added states 50 and 51 to check MAR before executing LDB/LDW/STB/STW instructions, and the vector is loaded with indices x02 and x03, respectively. If the exception is asserted then the rest of the execution is moved to state 49 which will handle the exception, otherwise the next state will be state 53 which points the the appropriate load or store instruction based on a logic done on the current opcode (logic on microsequencer diagram). Similarly, state 40, 57, and 48 are added to check the PC for any protection or unaligned access exception, and if there's no problem, state 48 just resumes the normal flow by pointing to state 33.
Note: states 57, 48, 51 and 53 have only one bit difference with state 49 and this is needed so the microsequencer can properly work.

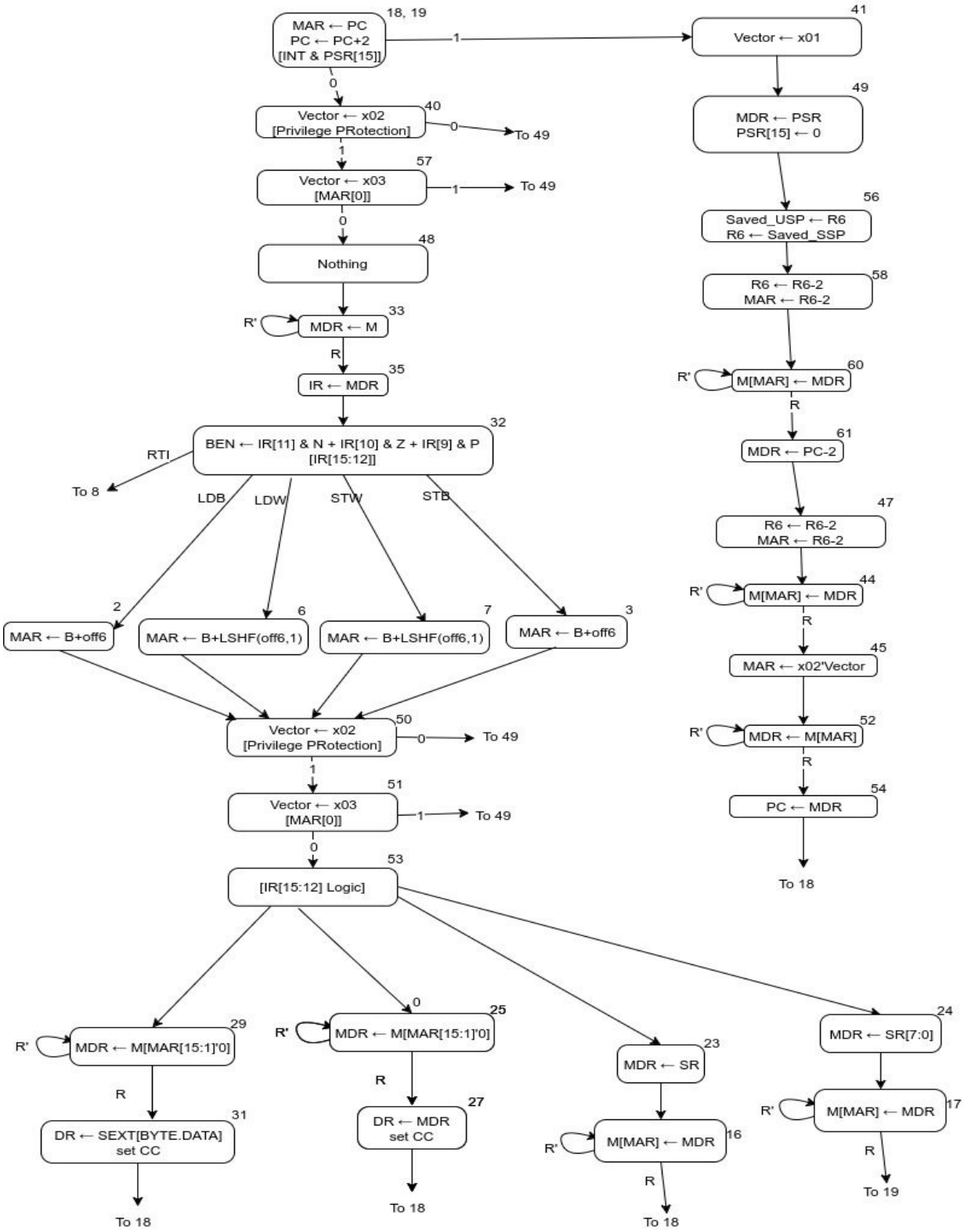

Figure 1. Interrupt and Exception flow

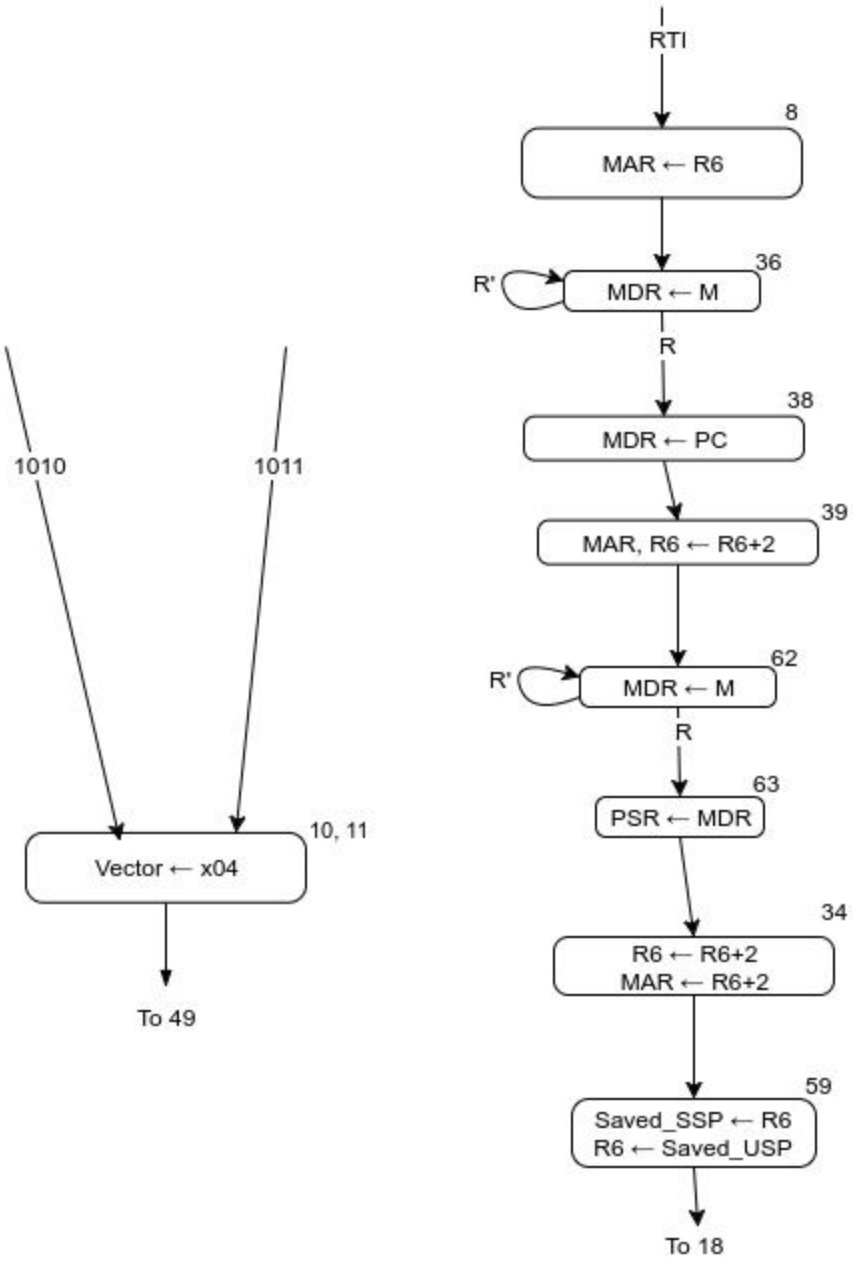

Figure 2. Opcode 10 and 11 and RTI

2. Changes you made to the datapath. Clearly show the new structures added, along with the control signals controlling those structures.

First, I added a Vector register with a LD.Vector load signal and connected it to a tristate with GateVector so I can load its value on the bus. The register feeds from a SignEXTend and

LeftSHiFt logic box and it gets its values from a MUX that can produce x01, x02, x03, or x04 depending on the value of VectorMUX.

Second, I added another MUX (controlled by SPMUX) that takes a value from Saved_SSP register, Saved_USP register, R6+2, or R6-2. Saved_SSP register, Saved_USP register, +2, and -2 registers are all connected to REG FILE and they will get their value from R6, stack pointer. DR1MUX is modified to pass IR[11:9], R7, or R6. Saved_SSP register and Saved_USP register both have their own load signals. Finally, the MUX (SPMUX) is connected to the BUS and is guarded by GateSP.

Last, I added a Priv register that stores the current privilege mode, controlled by LD.Priv, and its value can be fed to CONTROL or the BUS, which is guarded by GatePSR. NZP bits now can be set through a logic module or from the BUS (i.e. PSR[2:0]) depending the the PSRMUX. NZP can now feed the BUS, which is guarded by GatePSR.

Note: PC-2 can also be fed to the BUS and is Gated by GatePC2.

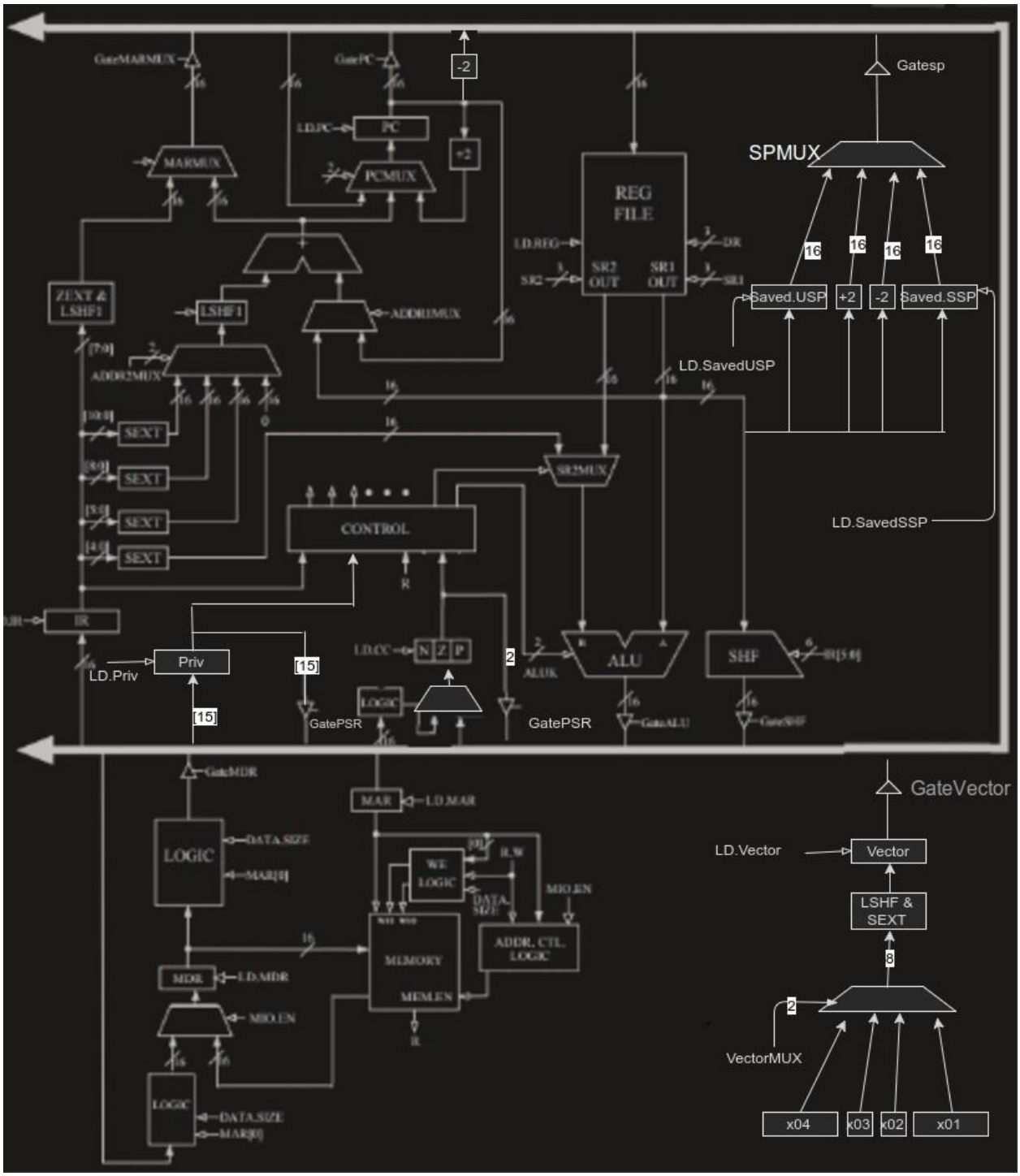

Figure 3. Data Path

3. New control signals you added to each microinstruction. Briefly explain what each control signal is used for.

Explanation provided in previous answer.

- LDST : returns the control back to either LDB, STB, LDW, or STW.
  - No, Yes
- GateSP : gate SP MUX
  - No, Yes
- GatePSR : gate Priv and NZP
  - No, Yes
- GateVector : gate Vector
  - No, Yes
- LD.Vector
  - No, Yes
- LD.Priv
  - No, Yes
- LD.SavedSSP
  - No, Yes
- SPMUX
  - Saved_SSP
  - -2
  - +2
  - Saved_USP
- VectorMUX
  - x01
  - x02
  - x03
  - x04
- LD.SavedUSP
  - No, Yes
- PSRMUX

- Logic
- BUS

- GatePC2
  - No, Yes

4. Changes you made to the microsequencer. Draw a logic diagram of your new microsequencer and describe why you made the changes.

I added a new condition variable (COND2). In addition to Address Mode, Ready, and Branch checks, I have added the logic for Privilege Protection, Interrupt Mode, and Unaligned Access. I have also added a new MUX that is controlled by LDST, which picks between 0,0,IR[15:12] or one of the states 29, 25, 23, 24 based on the logic given in the image below.

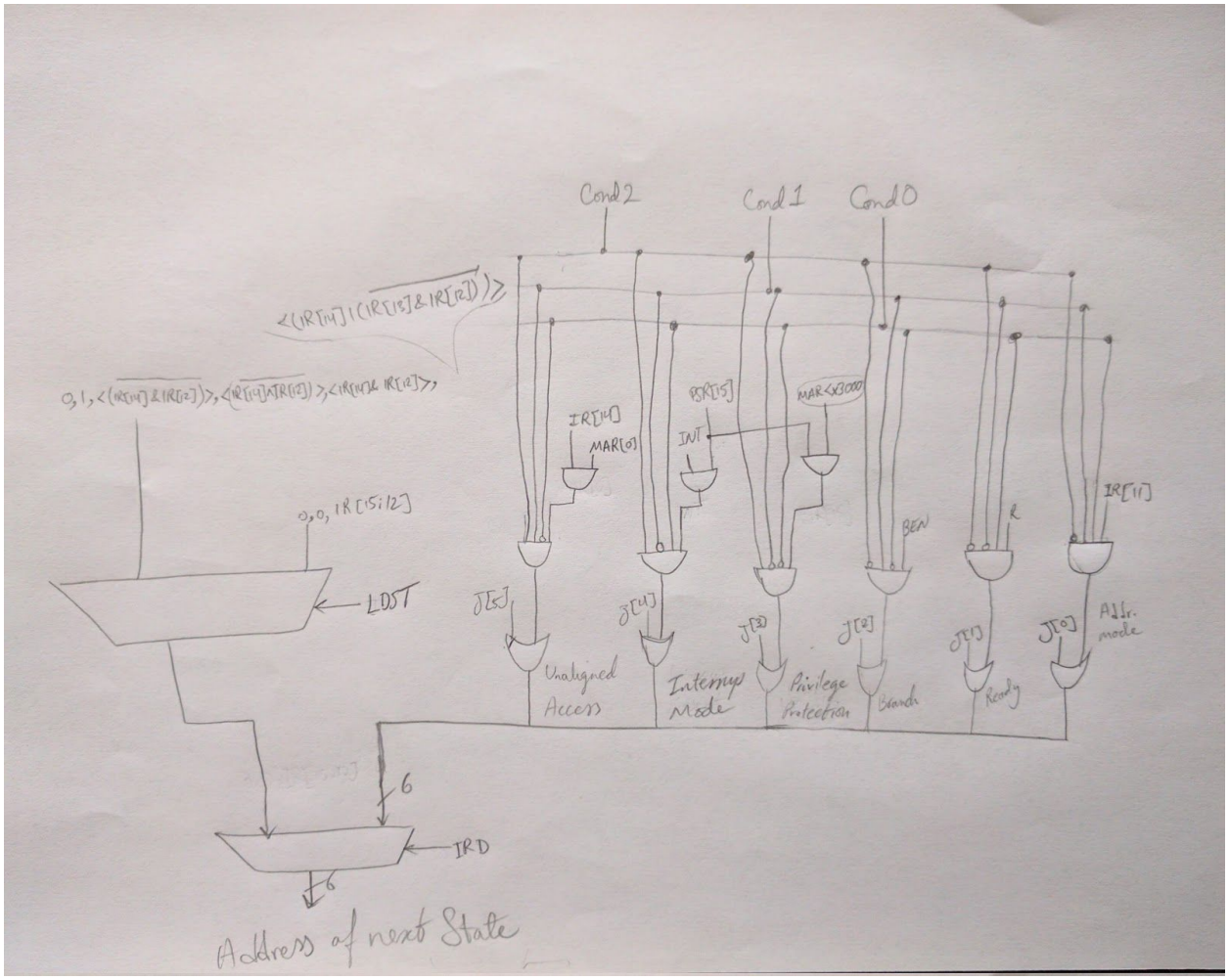

Figure 4. Microsequencer