**RISC-V Class Project Phase 8 – Data Memory Schematic + CA**

Phase 8 of the Class Project adds the logic to implement Data Memory load and store operations to the Schematic and the Codasip implementation created in Phase 8. The schematic update will be described first, followed by the Codasip code update.

1. **Update the Schematic**

Copy the schematic standardname7.xml to standardname8.xml. Alternatively named schematics will not be accepted. Open the schematic in draw.io.

The Data Memory is shown in Figure 4.49 in the textbook, reproduced in Figure 1 below with the Data Memory highlighted. The Codasip memory model requires a few changes to the textbook block diagram.

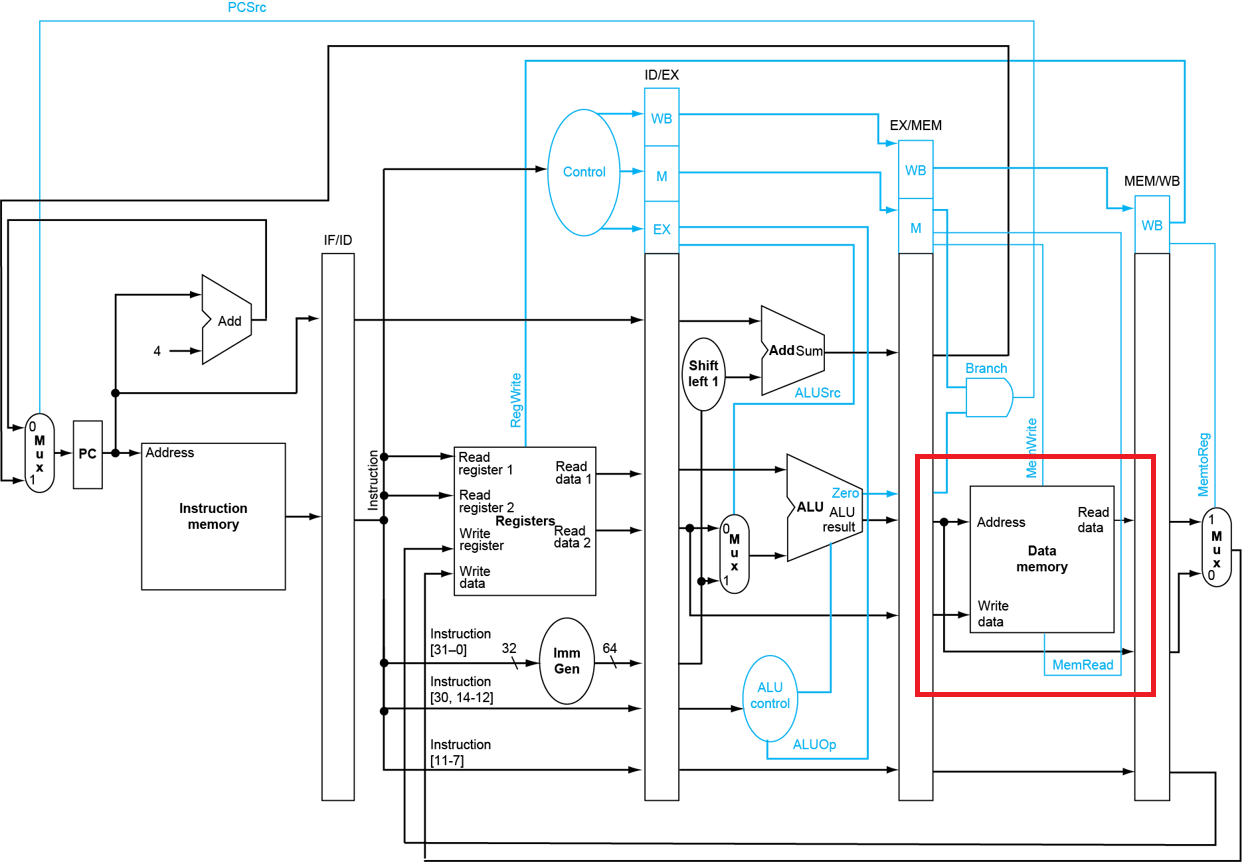


Figure 1

* 1. **Add the Memory Address Block and MEMCTL**

The Codasip memory model is split into two pieces, the address component and the data component (just as the Instruction Memory was in Phase 4). Add the address component to the EX stage.

The memory model requires a single OP value which includes both the read and write enables (the MemWrite and MemRead signals in Figure 1) and other information such as the number of bytes to be read or written. Create a new Decoder output s\_id\_memop which includes all of this information and pipeline it to the right stages. The address to the Data Memory comes from the ALU output before it is pipelined to the ME stage, but after the control block described below. The STAT output can be left unconnected for now. Add a label above the address block which is if\_data.

We will add a special function to control the access time of the Data Memory in order to simulate delays in accessing the memory. This function is specific to Codasip and is not described in the textbook. Add a Control block called MEMCTL. This will receive six inputs (the memory operation, the memory write data in, the memory address in, the memory write data from the ME stage pipeline, the memory address from the ME stage pipeline and a counter r\_me\_memcnt) and create new five signals. One of these creates the memory operation (s\_ex\_memop) to the actual Memory Address block. A second creates the input to a counter register (s\_ex\_memcnt) in the ME pipeline. The third creates a modified version of the memory write data (s\_ex\_wtdat) which is also pipelined to ME. The fourth creates a modified version of the memory address (s\_ex\_aluaddr) which is also pipelined to ME. The final one is called s\_ex\_resp, which will be used as the memory active/idle indication.

The memory operation will also be used in the WB stage, so pipeline the modified version there.

* 1. **Add the Memory Data Block**

Add the Codasip memory data component in the ME stage. The write data for a store normally comes from the output of the Register File selected by the rs2 field, but Figure 1 does not include the forwarding logic or the MEMCTL block. Pipeline the MEMCTL write data signal to the ME stage as r\_me\_wtdat and connect it to the WDAT input of the memory data component. Add a label above the data block which is if\_data.

The read data output from RDAT is pipelined to the WB stage as r\_wb\_memdat.

* 1. **Add Sign Extension to the Data**

The Codasip memory model produces either 8, 16 or 32-bit data as a function of the OP, with the selected data in the lower bits of the result. This is the same basic behavior as the standard RISC-V memory, but RISC-V includes some instructions which are signed loads requiring sign extension to a full 32-bit result. In the WB stage, add a control block SIGNEX which receives the pipelined read data and a pipelined version of s\_id\_memop, and produces the final memory read data s\_wb\_memdat. This is connected to the third input of the 3-1 mux added in Phase 7.

* 1. **Implement Load Data Hazard Handling**

As described in section 4.7 of the textbook, a Data Hazard can occur on a load instruction which cannot be resolved by the forwarding logic we have implemented. This requires the creation of a stall in the pipeline. Add a control component named LDHAZ in the ID stage which creates the signal s\_id\_loadhaz. Connect the required inputs, noting that MemRead is included in the s\_id\_memop control signal. The signal s\_id\_loadhaz will be used in the stall logic in Codasip which is not shown in the schematics.

1. **Examples**

Figure 2 shows some rough sketches of what the schematic pages 2 through 5 should look like.

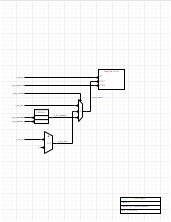
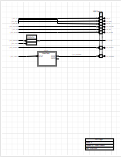
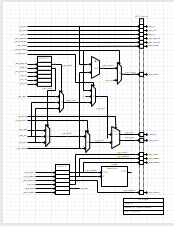
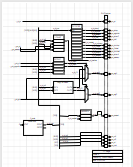


Figure 2

1. **Submit the Updated Schematic**

Although not required, it is recommended that you submit the updated schematic as soon as it is complete, as standardname8.xml. This will allow for feedback prior to creating the Codasip code. The scoring will reward early submission of the schematic. Submit to the Professor in Slack.

1. **The Function MEMCTL**

The control block MEMCTL is included to add a delay to the accesses to the Data Memory, which will be valuable in Phase 10. This function is NOT described in the textboook and is NOT a standard part of a RISC-V processor, although a similar function often exists in most memory interfaces. It is included to provide an exercise in implementing state machines.

The MEMCTL block receives the same inputs that the memory address block would (the address and the memory operation) and includes a counter, which is initialized to zero. If no memory operation is detected (the r\_ex\_memop signal is MEM\_NOP) MEMCTL simply passes the operation and address through unmodified (although they are not used) and the counter remains at zero. If a memory operation is detected and the counter is 0, the memory operation out of MEMCTL is changed to MEM\_NOP, the counter is incremented and the response signal s\_ex\_resp is set to RESP\_WAIT, which will stall the EX, ID and IF stages of the pipeline. The same operation continues until the counter reaches MAXCNT (a constant defined in ca\_defines.hcodal which should be set to 4). At that point the memory operation is set to the actual input operation (so that the memory reference is initiated), the counter is set back to zero and the response is changed back to RESP\_ACK which releases the pipeline stall.

The memory address and write data must be saved at the start of any memory operation, so if the counter is 0 the address and write data outputs are set to the normal input values. If the counter is not zero, the address and write data outputs are set to the saved value which is held in the ME pipeline. This means that when the counter reaches MAXCNT and a real memory operation occurs, the address and write data are presented correctly.

Signal values are “held” by defining a register for them in the ME pipeline stage (which is never stalled). The outputs of these registers are fed back as inputs to MEMCTL so that it can select either the input value or the stored value. Since MEMCTL controls the inputs to those registers, it can determine whether to store new values or retain the current values.

1. **Update the Codasip Files**

Changes will be made to several of the stages, in most cases requiring updates to ca\_resources.codal and ca\_defines.hcodal. Copy your standardname7 project to standardname8.

* 1. **Stalling**

Phase 8 will implement several stall operations. To prepare for this, in the IF, ID, EX and WB stages assign the local stall signal (e.g. s\_me\_stall) to the stall signal from the stage to the right (e.g. s\_wb\_stall). This will allow us to stall one stage and insure that all stages to the left are also stalled, which is necessary to prevent the loss of an instruction. All of the stall signals were created in ca\_resources.codal in Phase5\_orig. Several stall assignments will be modified in subsequent sections below.

* 1. **ca/include/ca\_defines.hcodal**

Create the enum defines for the memory operation control signal. Since this is used in a number of different places, create a separate enum for each memory instruction as shown in Figure 3, with a width of MEMOP\_W.

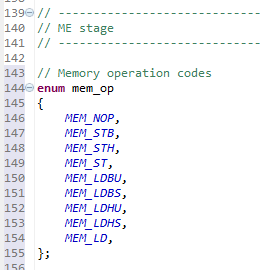


Figure 3

Add the third choice WBMX\_MEM (or an appropriate name based on the other enums) for the Register File write mux in the WB stage.

Create a new enum for the response signal from the MEMCTL block, with choices named RESP\_ACK and RESP\_WAIT.

Add a #define at the end called MAXCNT, and define it to be 4. This defines how many cycles we wait for the Data Memory access, and we will modify this in Phase 10. Create a width define for it, using bitsizeof(MAXCNT) since there isn’t an enum for this.

* 1. **ca/resources/ca\_resources.codal**

Add the new DECODER control signals and the pipelined versions.

Add the Data Memory read and write data signals and the pipelined versions, and the response signal.

Add any new signals required by MEMCTL, including the counter.

Add the sign extension data signals.

Add the load hazard signal in the ID stage.

* 1. **ca/pipelines/ca\_pipe\_stage3\_ex.codal**

Add the address part of the Data Memory. The structure of memory is fairly complex, so create a new event ex\_memory which looks like ex\_output. Call ex\_memory just before calling ex\_output in the main semantic section. The beginning of the memory portion of ex\_memory is shown in Figure 4. The variable definitions are needed for creating the control signals to the memory block. The 32-bit address to the memory block is s\_ex\_aluaddr (the modified address), since the RISC-V memory address is created from the sum of a register and an immediate in the ALU. “BUS\_MASK” is a constant with ones in the lower 2 bits and zeroes otherwise (the value 0x3), so ANDing that with the address produces the byte offset in byte\_offset, and ANDing the complement of that produces the word aligned address base\_address.

Each value of s\_ex\_memop selects two values – the command request\_cmd which is either CP\_CMD\_NONE, CP\_CMD\_READ or CP\_CMD\_WRITE, and byte\_count which selects the number of bytes to read or write. Add the entries for all of the other possible s\_ex\_memop enums from Figure 3.

The event ex\_memctl implements the MEMCTL control block. It must be called before the rest of the memory code to avoid forward references. The “use” statement is required.

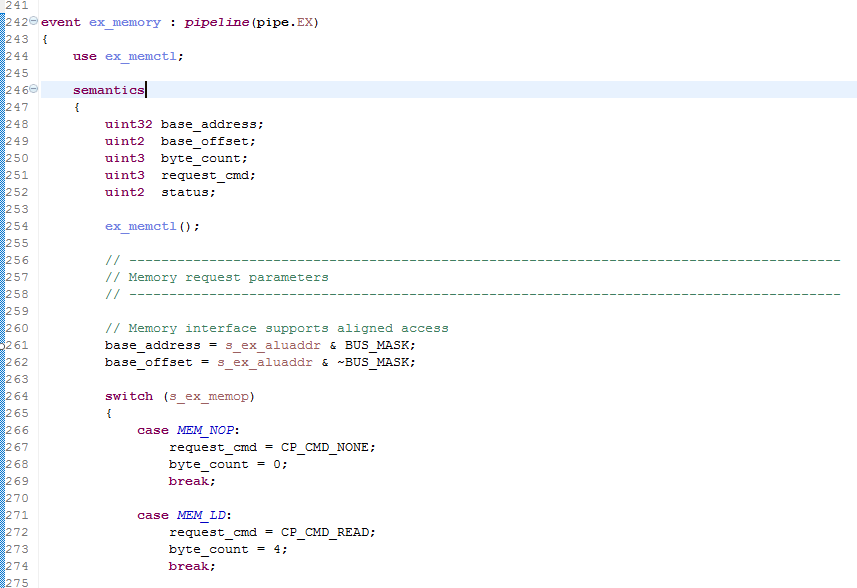


Figure 4

Once the parameters have been calculated, instantiate the actual address block just as was done for the Instruction Memory in the IF stage in Phase 5. Figure 5 shows the code for this. Note that the variable “status” was created to receive the STAT output of the memory block although it will not be used in this phase.

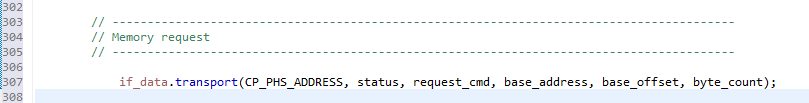


Figure 5

Add any appropriate pipeline registers to ex\_output.

Add the function block to implement MEMCTL. This will be a set of statements which create each of the outputs based on the input signals. The three values of the memory counter, the write memory data and the memory address are all held in registers in the ME stage pipeline and are fed back to MEMCTL as inputs

MEMCTL has an output s\_ex\_resp which will generate an EX stall when its value is RESP\_WAIT (one of the enums). OR this into s\_ex\_stall, which is normally just s\_me\_stall.

* 1. **ca/pipelines/ca\_pipe\_stage4\_me.codal**

As in the EX stage, create a me\_memory() event and call it prior to me\_output(). This function is shown in Figure 6. The memory data function returns the read data s\_me\_memdat. The fourth parameter is the write data value. s\_me\_resp is the response signal.

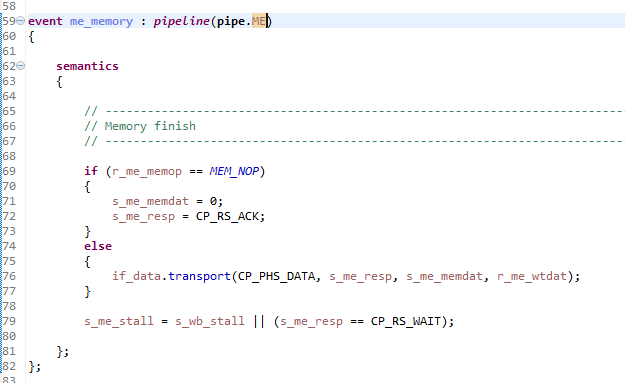


Figure 6

If the memory operation is not complete (s\_me\_response is CP\_RS\_WAIT), the pipelined must be stalled. If any stage of the pipeline is stalled, all of the stages to the left of it must also be stalled. This means that the ME stage stall signal s\_me\_stall is asserted if either s\_wb\_stall is asserted in the WB stage OR if a stall is required because of the memory operation.

Add the appropriate pipeline registers.

* 1. **ca/pipelines/ca\_pipe\_stage5\_wb.codal**

Add sign extension for the read data which is pipelined to the WB stage. One example is shown in Figure 7. Add all of the remaining read choices. Add the output of the sign extension function as the third input to the Register File write data mux.

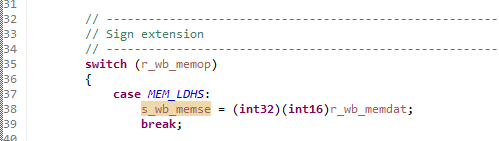


Figure 7

* 1. **ca/pipelines/ca\_pipe\_stage1\_if.codal**

The IF requires some special functions to handle stalling since some registers cannot be handled with the automatic stall function in pipeline\_control. r\_pc is not part of a normal pipeline, so add another conditional to the selection of the next PC value which keeps the current PC value if s\_if\_stall is asserted.

The address register within the Address part of the Instruction Memory also cannot be stalled. In order to make sure the Instruction Memory data output is held on a stall, replace the single if\_code.transport function with the expanded code in Figure 8. Since r\_id\_pc holds the previous value of r\_pc, this structure will hold the value on the memory output.

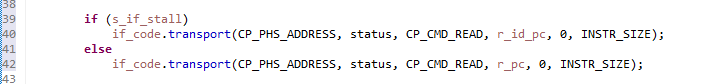


Figure 8

s\_if\_stall should have been assigned to s\_id\_stall. Make sure to avoid forward references with s\_if\_stall.

* 1. **ca/pipelines/ca\_pipe\_stage2\_id.codal**

Add the logic of the control block LDHAZ to create the signal s\_id\_loadhaz.

The detection of a load hazard inserts a bubble into the pipeline, as described in section 4.7 of the textbook. This is implemented by having s\_id\_loadhaz assert (i.e. OR a 1 into them) the correct stall and/or clear signals. In each case this should be added to the stall or clear signal in the stage where that signal is created. Because the stall function is propagated to the left in the pipeline as described in section 5.5 above, only the rightmost version of stall should be affected.

* 1. **ca/decoders/ca\_decoder.codal**

Add any new control signals from the DECODER to every existing instruction group with the appropriate value.

Copy one of the existing instruction groups to a new group i\_hw\_load corresponding to the i\_load group in isa.codal and change the control signals to the correct values. There will be a switch statement which should have a default, so create a new MEM\_ERROR code in debug.hcodal and use it there.

Copy the i\_hw\_load group to a new group i\_hw\_store corresponding to the i\_store group in isa.codal and change the control signals to the correct values. Use MEM\_ERROR in the switch statement default.

* 1. **ca/other/ca\_utils.codal**

Remove all of the comments from ca\_utils.codal to enable all of the - -info arguments as shown in Figure 9. - -info 8 enables the ADDR print (the memory address) and - -info 9 enables the MEMDAT print (the memory read and write data data). These are valuable in debugging the MEMCTL behavior. Make sure all of the variables referenced by the prints are defined, as your names may be somewhat different.

A screenshot of a cell phone

Description automatically generated

Figure 9

1. **Build the Project**

Once all of the CA files are updated, build the project by first double clicking the button to the left of Model Compilation (ca) (NOT Model Compilation (ia)) in the Task window of the Codasip Perspective. Any build errors will appear in the Console window, so correct any missing assignments, syntax errors, etc. Continue until the Model Compilation builds correctly. The previous if\_data warning should now be gone, so there should be no warnings.

Build Simulator (ca) in the Task window, and again correct any syntax errors.

1. **Run the Test Program**

Once both the Model Compilation (ca) and Simulator (ca) tasks finish successfully, the next step is to run the test program phase8\_test. Import this test program and build it with your Phase 5/6/7 IA SDK.

Execute Run -> Debug Configurations as in Phases 1/2/3. The C/C++ Project should be phase8\_test, and the Application should be Debug/phase8\_test.xexe. For the debugger, select standardname8.ca.standardname8-ca-simulator. If this is not a choice, the project build process has not completed successfully.

The running and stepping controls are the same as described in Phase 2. The debugging functions described there will be useful. The test successfully passes if the value in register x25 is 1 and the value in register x10 is 57, after executing the run (the green arrow) exactly one time.

Additionally, when the test completes the simulation must have run for 561 cycles, as shown in Figure 10. Other values indicate an error in the design of MEMCTL.

A screenshot of a cell phone

Description automatically generated

Figure 10

1. **Scoring the Project**

The project should be submitted when the test program phase8\_test passes when using the CA Model (be sure to select this in Debug Configurations and not the IA model which was used in Phases 2 and 3). Unsuccessful submissions will be rejected. The hardware project counts for 80% of the total Phase score.

The score for a successful Project submission will be determined by the time of submission relative to the Target Date. A bonus of 1% of the Project score (i.e. 0.8% of the Phase 8 score) will be added for each day earlier than that the successful project is submitted. A penalty of 4% of the Project score (i.e. 3.2% of the Phase 8 score) per day will be imposed after the Target Date.

The score for the schematic will be determined by the time of submission with deductions for any errors. Error deductions will be similar to those for Phase 4 (1 point for a bad signal name, 2 points for a bad connection, etc.) but the deduction will be from the entire Phase 8 score, not just the 20% which is for the schematic. A bonus of 1% of the schematic score (i.e. 0.2% of the Phase 8 score) will be added for each day earlier than the Target Date a fully correct schematic is received, up to a maximum of 7%. A penalty of 4% of the Schematic score per day (i.e. 0.8% of the Phase 8 score) will be imposed after the Target Date. Submitting the schematic early not only gains schematic bonus points but produces a correct schematic which aids in creating the actual hardware project.

1. **Exporting the Project**

Once the test program is running, the project should be Exported as standardname8.