Implementing Tepid

For the remainder of the semester, our lab work will focus on the implementation of the Tepid architecture as a circuit diagram constructed using Logisim. Your primary source of information as you work on this project will be the separate "Tom's Eviscerated Processor Instruction set Design" handout which describes Tepid. Note that the description of Tepid in that handout has been substantially revised since the first draft of the handout was distributed before Lab 8. Several features have been removed from the architecture described in the first draft and the descriptions of other features have been clarified or corrected. Make sure that you have an updated version of the Tepid handout (dated November 13th or later).

We will continue to hold labs on Tuesday afternoons to provide us a chance to guide you as you complete your projects. In addition, as described below there will be intermediate products due at various points to ensure that you are making progress on the project.

As mentioned in class. You are **strongly** encouraged to work in pairs on this project. Each pair will only need to submit one copy of each product required during the project. Those who don't work in pairs will be considered anti-social. You need to commit to working as a pair (or not) by the next lab meeting.

A Shrinking Architecture

The most important revisions to the Tepid architecture involve the removal of features. While we have maintained enough features to make it possible to write interesting programs in Tepid, we have done our best to remove unnecessary extras.

Go-to's Considered Harmful?

The main feature eliminated from the original design is the branch instruction format and the two instructions, branch and branch and link, that went with it. One of the interesting features of Tepid is that the program counter is accessible as a general purpose register (r15 or pc). This makes it possible to use other instructions in place of branches in many situations. For example, the instructions

```
b      loopStart
```

and

```
adr    pc,loopStart
```

both cause unconditional branches to the instruction labeled loopEnd. Similarly,

```
ble     loopEnd
```

and

```
adrle    pc,loopEnd
```

branch to loopEnd only if the last operation that set the condition code produced a value less than or equal to 0.

There is a major difference between using b and adr to update the program counter. The adr instruction is encoded as a type 1 (load/store) instruction which computes the address of the second operand using a base register (which can be the pc) and a 14 bit signed displacement. This means an instruction like

```
adr    pc,loopStart
```

will only be accepted by the assembler if the address of the instruction labeled loopEnd is within $2^{13} = 2048$ words/instructions of the adr instruction. The instruction

```
b    loopStart
```

on the other hand, uses a 24 bit offset, supporting a much wider range of branches. We strongly doubt that any of you will write test programs containing more than 2048 instructions, so we do not expect this issue to be a limitation in practice.

It is also possible to replace the single branch and link instruction

```
bl    someFunction
```

with the two instruction sequence

```
add   lr,pc,#2
adr   pc,someFunction
```

The add instruction saves the address of the instruction after the adr instruction in the link register (r14), then the adr instruction transfers to the first instruction of the function.

The elimination of branch and branch and link completely eliminates the need for the instruction format that goes with them. We hope you will appreciate this simplification (even though you might eventually realize that it might be even better if we had left the branch instructions in and taken the program counter out of the general register set instead!).

Un-conditional

A second important simplification involves the setting of the condition code. In Warm (and our original Tepid design), most instructions (including memory access instructions) could set the condition code register bits. In the revised Tepid design, only type 0 instructions (arithmetic and logical operations) can set the condition code. The obvious place to compute new condition code values is in the ALU. This simplification makes it unnecessary to run values going to or from memory through the ALU.

A Last Minute Deletion

In the hour between when I printed what I thought was the final, revised description of Tepid and the printing of this handout, James convinced me to drop two more instructions. Although they are included in the Tepid handouts I will distribute, you do not need to implement the cmp and tst instructions. Any use of a cmp or tst can be replaced with a subs or ands (as long as you have a free register to use as the target of the new instruction).

Sub-goals

As mentioned above, we would like to structure this project in a way that encourages you to make (and demonstrate) meaningful progress each week until the final deadline. With this in mind, we propose the following schedule:

By 11/20 Dataflow paths and control states designed. THIS WILL BE HARD! Start early.

By 11/28 Subcomponent implementations complete (except control).

By 12/3 Control circuitry complete (nothing but debugging left!).

By 12/6 Complete projects due.

Subcomponents

To guide you in your design and to clarify the list of subgoals given above, we will describe several subcomponents/subcircuits you should plan on including in your project. Most of these will be familiar from the Logisim MIPS implementation presented in the book and in class, but some are specific to TEPID and all will need to be tailored in some way for the TEPID architecture.

Register bank This machine requires a set of 16 registers. The interface to these registers should be identical to that used in the Logisim MIPS circuit, except the inputs used to address registers (A1, A2, and A3) should be 4 bits wide instead of 5. There should be no special inputs or outputs to access the program counter.

Data/Instruction Memory The memory subcomponent for TEPID should be a little simpler than what we used for MIPS since TEPID memory is word addressable rather than byte addressable.

ALU The ALU for TEPID will differ from that used for MIPS in two ways. It will need to compute the N and V condition code signals (in addition to the Zero signal included in our MIPS ALU). You may also want to add an additional operation which ignores one of the ALU inputs and passes the second or its negation through as a result. This will be helpful for the mov and mvn instructions.

Shifter TEPID provides flexible options for shifting data. To support this you should construct a sub-circuit that takes one 32-bit data input and an 8-bit control input containing both the shift operation code and the shift amount as encoded in a TEPID type 0 instruction. This circuit should produce an appropriately shifted/rotated 32-bit result.

Instruction Decoder As we did for the MIPS implementation, you should construct a sub-circuit that takes a 32-bit instruction as input and produces outputs corresponding to all of the interesting sub-fields that might be present in either of the TEPID instruction formats.

Condition Checker An important subcomponent of your control circuitry will be a sub-circuit that takes as input the current values of the condition code registers and the first three bits of the current instruction and produces a single bit indicating whether the instruction should or should not be executed based on its input values.

Data Path and Control State Design

Your first sub-goal will be to prepare and submit a design for the data path you will use to implement TEPID together with a finite-state machine design that demonstrates how your data path will be used to implement TEPID.

The data path you construct will consist of the subcomponents described in the preceding section interconnected with one another and an appropriate collection of registers and multiplexers. By 11/20 we want you to submit a Logisim project including your data path design. To facilitate this, we will provide you a starter Logisim project containing template sub-circuits[1] for each of the components described above. You should use this starter project to build a new sub-circuit named "data flow" showing how you will interconnect all of the subcomponents to form your data flow path.

[1] This starter does not exist as I type this handout, but the plan is to give you some sub-circuits that are complete or nearly complete (the register bank?) and others with nothing but appropriate inputs and outputs (the shifter?).

Your implementation of Tepid will have to be based on a multi-cycle control circuit. Two aspects of the Tepid architecture require the use of such a design. First, unlike the version of MIPS used as the basis for the single-cycle design presented in the book, Tepid uses a single memory for data and instructions. As a result, executing a single instruction may require two memory accesses. This would not be possible in a single-cycle implementation. In addition, the fact that Tepid treats the program counter as a general purpose register means that the execution of a single instruction will often require reading and writing multiple registers. As we saw in the MIPS implementation, it is possible to support reading multiple registers in a single cycle. Writing multiple registers in a single cycle is trickier, since conflicts may arise if both writes involve the same target. Therefore, your register bank should only allow a singled register to be updated each time the clock ticks. You should handle instructions that update both the program counter and another register by performing the two updates in separate sub-cycles.

Given that a multi-cycle approach is unavoidable, we encourage you to mimic the multi-cycle design presented in the text by inserting extra registers that limit the maximum gate delay required for any sub-cycle. In particular, like the text, you should probably insert A and B registers, though you may find it better to think of them as holding the ALU inputs rather than the memory bank outputs. You certainly will want an instruction register. We also had a register to temporarily hold the updated program counter value in our design.

The viability of the data paths you design can only be judged by also designing a set of control states that use the existing data paths to execute all of the instructions included in the Tepid architecture. The text presents its control design using circles representing states and arrows representing possible transitions. Each circle contains text describing control signal settings.

We believe a more appropriate way for you to design your control system is to draw a diagram of circle and arrows where the circles contain pseudo-code describing the actions that will occur as the machine passes through the state. An example of such a diagram for the control system used in the text's multi-cycle MIPS implementation is shown in Figure 1. In the diagram, it is assumed that there is an implicit arrow/transition from each of the nodes without any explicit outgoing arrow to the node at the top.

We expect you to prepare and submit such a state diagram with the Logisim project containing your data path design on 11/20. Your diagram should be as neat and readable as the on in Fig. 1. If you can accomplish this by hand, that is acceptable. Otherwise, copies of the program we used to create our diagram, Omnigraffle, are available for use on the Macs in the back room of TCL 312.

Developing your data path and control diagrams will be an iterative and likely a frustrating process. You will sketch out the data path and then discover as you try to sketch out the state machine that some instruction requires a data path that you left out or takes extra cycles because some path is absent. You will then revise the data path and rework the state machine only to discover some new obstacle. This will repeat until you devise a data flow diagram and control system that are consistent and can implement all Tepid instructions.

Sub-circuit Construction

Your second deadline for this project is to submit a copy of your Logisim project including what you believe to be complete implementations of the components described in the "Subcomponents" section above. This should just involve completing the templates we provided in the starter project. While we do not expect you to have completely debugged these sub-circuits, you should perform as much "unit testing" as possible.

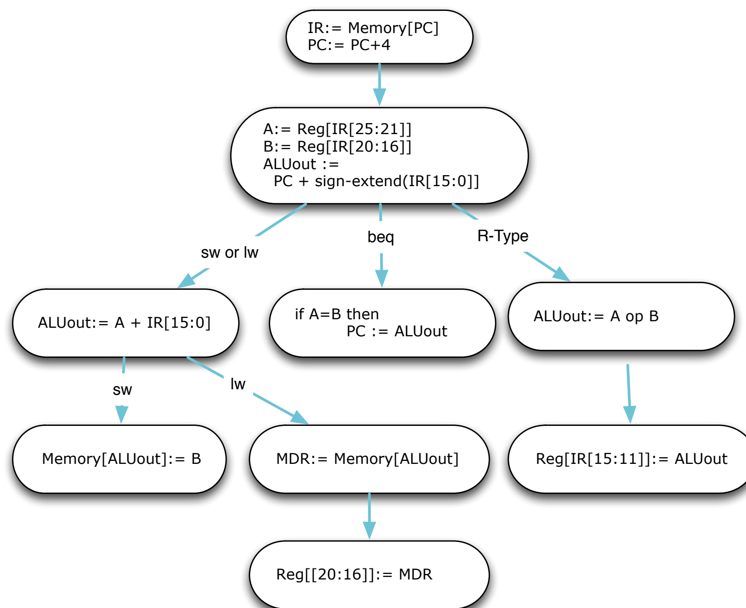

Figure 1: State diagram for multi-cycle MIPS control

Testing the Completed Circuit

Once you believe you have completed the data path, the control circuitry, and all of the sub-circuits, you will want to test your work by using the circuit to interpret TEPID programs. We have provided a few sample programs including an implementation of heap sort, but these are not appropriate for initial testing of your circuit. You should instead construct some very simple programs that contain just a few instructions and make sure these work correctly before trying a program that executes hundreds or thousands of instructions.

Since TEPID is a subset of WARM, you can develop test programs for your circuit using the tools Duane Bailey developed for WARM — waa and wai. That is, you should start by creating a text file containing your sample program. Then, you should assemble your program with waa to produce a .o file and run the program under wai to verify that it behaves as expected. If a program does not behave as expected when interpreted by your circuit, you want to be certain that the mistake is in your circuit rather than in the program. Testing the program with wai is therefore essential.

As you construct test programs, remember that TEPID is a strict subset of WARM. You will need to be careful not to use branch instructions or other features not included in TEPID as you write sample programs.

One particularly important feature of WARM that is missing in TEPID is the set of software interrupts (swi) that are used to do input/output and to terminate a program. When your circuit runs a TEPID program, it will not produce any output or consume any input.

In place of input, you may use immediate data values in instructions (indicated by the # symbol) or pre-initialized variables and arrays in memory (created using the assembler .data directive).

When debugging a test program that is eventually intended for your circuit with wai, it may be helpful to actually use swi to perform output at the beginning. Once the program appears to work, however, you should remove the code to produce output and verify that you can identify the desired results in the registers or memory of the machine using the debugger interface provided by

wai. When you switch to running the program with your circuit, you will used Logisim's interface to examine the same registers and memory locations to verify that your program was executed correctly.

Finally, where you would have used swi to halt your program, you should include an adr instruction that branches to itself:

```
adr pc, pc,#0
```

This creates a tight loop. It should be fairly easy to recognize when your circuit reaches this loop.

Once you are ready to test a program with your circuit, you need to convert the .o file produced by waa into a file in the format accepted by Logisim as a memory image. We have provided a script to do this conversion. If you have a file named test.o produced by was in your directory, then the command

```
tepidMem test.o > test.mem
```

will create a file named test.mem that can be loaded into Logisim[2].

As you saw in Lab 8, you can load such a file by

- Opening your data path circuit within your Logisim project.
- Click on the "data memory" component of the circuit. It should become highlighted and a magnifying glass icon should appear in the middle of its icon.
- Double click on the magnifying glass to see the details of the instruction memory.
- Control click on the ROM icon within the instruction memory sub-circuit.
- Select "Edit contents" from the menu that appears.
- Click on the "Open" button at the bottom of the window showing the memory values.
- Navigate to and open the .mem file you created using tepidMem.
- Make sure your code is now displayed at the beginning of memory.
- Close the memory contents window.

Submitting Your Work

When you are finished working on your Tepid implementation, you should use the turnin program to submit the following items for grading:

- A Logisim .circ file named tepid.circ containing all of the subcircuits used in your implementation.[3]
- A postscript file containing drawings of your subcircuits created by directing the output of a Logisim printout of all of your subcircuits to a file named tepid.ps. To produce this file:
  - Select the "Print" item from the Logisim File menu. A dialog box should appear.

[2] Assuming you have previously executed a source command to use the 237 Linux environment.

[3] Please make sure that you delete any subcircuits you might have created that are not required by your final implementation before you submit this file or print drawing of your circuits!

  - Shift click or drag to select the names of all of your circuits in the list in this window.
  - Edit the "Header" field in the window by adding your name(s). Click OK. A new dialog box should appear. Make sure the Print to File check box is selected. Then click Print.
  - Use the next dialog box to navigate into the folder where your .circ file is saved and save the file containing drawings of your circuits as tepid.ps in this folder.

- At least one .s file and an accompanying .mem file containing a test program you created to verify the correctness of your implementation (i.e., not heap sorts).
- A README-tepid file containing:
  - Your name(s)!
  - Brief descriptions of the test program(s) you submitted including a short description of how to run them and verify that they behaved as expected.
  - A summary of your assessment of the correctness of your implementation. That is, briefly describe the range of test program that your implementation has run correctly and also describe any programs that do not run correctly (including any theory you might have about what is wrong).

Finally, you should also submit a diagram describe the finite state machine implemented by your control circuitry. This diagram can be created with a drawing program or hand drawn but it must be clear and legible. Please use the style illustrated in Figure 1 (where each bubble contains pseudo-code) rather than the text's style (where each bubble contains control line settings). You can either submit this drawing using turnin under the name tepid-control.pdf or you can just turn it in on paper by class on Friday