EE 361L  Fall 2010
Pipelined MIPS-L0 (PMIPS-L0) and Pipelined MIPS-L (PMIPS-L)

Last updated:  November 8, 2010

1. Introduction

PMIPS-L0 and PMIPS-L are pipelined versions of MIPSL (for MIPS Lite). Appendix A has a description of the MIPSL processor. PMIPS-L0 and PMIPS-L have fewer instructions. These instructions are R-type, addi, lw, sw, and beq. The other instructions (j, jal and jr) are not implemented.

PMIPS-L0 and PMIPS-L have five pipeline stages just as the pipelined MIPS in Chapter 4 in the textbook as shown in Figure 1. The PMIPS-L0 only allows one instruction to be processed at a time. This means that during the execution of an instruction, the instruction fetch (IF) stage must stall and the controller must insert bubbles into the pipeline. This is explained in Section 2. Thus, PMIPS-L0 doesn’t really pipeline its instructions.

PMIPS-L does some pipelining as described in Section 3. Parts of the datapath are the same as in the single cycle MIPS except the controller, pipeline stage registers, and perhaps a modification to the PC logic.

All the sequential components are synchronized with the positive clock edge except the register file, which is synchronized with the negative clock edge.
2. PMIPS-L0

For this processor, only one instruction is being processed at a time, each instruction taking exactly four clock cycles. The following are the states:

- State 0, Instruction Fetch: PC = PC+2, and insert Bubble into ID/EX stage.
- State 1, Instruction Decode: PC holds value, and the controller sets control signals to ID/EX stage that is dependent on the opcode in IF/ID.
- State 2, Bubble: PC holds its value, and insert Bubble into ID/EX stage.
- State 3, Bubble: PC holds its value, and insert Bubble into ID/EX stage.

Figure 3 shows an example of beq progressing through the computer. Note that at State 2, the beq instruction is at the EX/MEM pipeline register. At this point, PCSrc (= EXMEMBranch & EXMEMALUZero) has the outcome of the condition of beq, i.e., PCSrc = 1 means the computer should branch. If PCSrc = 1 then this will cause the PC to load in the target branch address of beq at State 3.

**Figure 3.** Progression of beq instruction in MIPS-L0. Here “bub” is a bubble, and “NI” is next instruction after beq.

PC Logic: Note that the PC logic must be able to load the target branch address into the PC while the PC is in the stalled state. Think about how you can modify the PC logic by adding a 2:1 multiplexer. A solution is provided in Appendix B.

Controller: This is a sequential circuit with four states. It repeatedly goes through states 0, 1, 2, 3, 0, .... The controller inputs the opcode of the instruction from the IF/ID register.

When the controller inserts a bubble, it must disable all the control lines. The critical control lines to disable are the ones that can change state values in registers or memory. These registers and memory are located in the register file,
program counter and data memory. Thus, to insert a bubble means that the register-write and memory-write must be disabled. We must also prevent the PC from being loaded inadvertently. Recall that during a stall, the controller will set the PC to a stalled state. Then the only way the PC can be inadvertently set to the wrong value is due to an inadvertent branch. To prevent this, set Branch=0.

Pipeline Registers: These registers are basically just a bank of D flip flops. There should be enough flip flops to load all the critical signal information.

As an example, the ID/EX register has the following fields:

- **WB**: \{RegWrite,MemtoReg\}
- **M**: \{MemRead,MemWrite,Branch\}
- **EX**: \{ALUSrc,ALUOp,RegDst\}
- 16-bit output of the sign extension
- Outputs of the read data from the register file
- Two register fields (5-bits each) from the instruction

There are two memories: instruction memory that has the program, and data memory which has RAM.

For EE 361L lab 5, you will implement PMIPS-L0 in an FPGA. Also, your computer will need Input/Output (IO). The IO will be connected to other circuits on the Digilent Basys boards. In particular, the IO of your computer will be attached to a seven segment delay and sliding switches. Your computer will need two IO ports, one for the display and the other for the switches. The port for the seven segment display is 0xfffa, and the port for the sliding switches is 0xfff0. These addresses are in the address space of data memory. So you will access the IO using the lw and sw instructions. The seven segment display can be written to, and the sliding switches can be read from. The sliding switches are labeled SW1 and SW0 and are the last two bits of the data read from 0xfff0.

The RAM portion of the data memory has address 64 up to 126. Note that since the memory is byte-addressable and each memory word is now 16 bits (= 2 bytes), then each memory word has addresses divisible by two. The reason why the RAM has addresses from 64 to 126 rather than 0 to 62 is due to the fact that I’m reusing this memory module that was used in another project. Anyway, it’s good practice to work with memory that’s at an odd location since in real systems, memory does not always start from address 0.
3. PMIPS-L

We will pipeline the instructions further but avoid any data or control hazards. The processor will have simple branch prediction, and in particular branch-untaken (which means it predicts that the branch will not be taken). Here are the two parts of the processor to modify.

Simple Branch Prediction, Branch-Untaken: We introduce an additional bit into the IF/ID pipeline register called Valid, e.g., call it IFIDValid. It indicates if the instruction in the register is valid. If it is invalid then the controller assumes it is a nop. By default, IFIDValid = 1.

When a branch is executed at the EX stage, it can cause PCSrc to become 1. This means that the computer will jump to the target branch address, i.e., PC = target branch address. Then all the instructions in the pipeline should be flushed out as follows:

Put bubbles or clear the pipeline registers ID/EX and EX/MEM, and invalidate the value in the IF/ID register by setting IFIDValid = 0.

The following figure illustrates what happens when a branch is taken.

- **Cycle 1**: Target Branch Address, beq, Bubble, invalid.
- **Cycle 2**: Target Branch Address, beq, Bubble, invalid.

Pipelining and Avoiding Data Hazards:

There are two types of data hazards, those that deal with register file values and those that deal with data memory values. Due to the load/store architecture of the 5-stage MIPS, there are no data hazards through data memory. Thus, we will only concern ourselves with data hazards through the register file. An example is as follows:

- `add $1,$2,$3`
- `sw $4,10($1)`
- `addi $5,$1,37`

Note that “addi” and “sw” are dependent on “add” via register $1$. Thus, the “addi” and “sw” should be prevented from entering the pipeline until $1$ is properly updated.
Instructions that write to a register are labeled *data-hazard instructions*. The register that they write to will be referred to as the *destination register*. Among the five instructions (addi, Rtype, lw, sw, and beq), the data-hazard instructions are addi, Rtype, and lw.

One way to avoid data hazards is to prevent instructions in the IF/ID register from entering the pipeline if there is a data hazard due to a data-hazard instruction already downstream in the pipeline. For example, consider the following two instructions.

\[
\text{add} \quad \$1, \$2, \$3 \\
\text{sw} \quad \$4, 10(\$1)
\]

Note that “sw” is dependent on “add” via register $1$. Now suppose “sw” is at the IF/ID pipeline register. If “add” is at the ID/EX or EX/MEM pipeline registers then we should stall “sw” from entering into the processor pipeline, otherwise there will be an error. However, if “add” is in the MEM/WB then the new value of $1$ will be written back into the register file on the negative clock edge, and will be available for “sw”. Thus, “sw” can be sent into the pipeline processor.

More generally, if there is an instruction A in the IF/ID pipeline register, it should be prevented from entering the pipeline processor if

- There is a data-hazard instruction B at ID/EX or EX/MEM and the destination register for B is one of the source registers of A.

To keep track of all this, the pipeline registers ID/EX and EX/MEM must be expanded. In particular,

- ID/EX needs new registers
  - IDEXDataHazard, which is a single bit indicating whether the instruction at this stage is a data-hazard instruction
  - IDEXDstReg, which is a 3-bit value that is the destination register of the instruction.
- Similarly, EX/MEM needs new registers EXMEMDataHazard and EXMEMDstReg.

To prevent an instruction from entering the pipeline from IF/ID, the controller must the keep the PC from incrementing by 2 (however, it should allow a branch from a downstream pipeline stage), and to insert a bubble into the ID/EX pipeline register. Note that a bubble implies setting IDEXDataHazard = 0 because it is a nop and cannot cause a data error.

Otherwise, the controller should increment the PC by 2 and send the control signals of the instruction in IF/ID into ID/EX.

Here is an algorithm for the controller to follow. It has two levels of processing. The first level is to identify all the registers that should be compared to detect a data hazard.

Let IDEXRegCompare be a 4-bit value where

\[
\text{if (IDEXDataHazard} == 1 \&\& \text{IDEXDstReg} != 0) \text{IDEXRegCompare} = \text{IDEXDstReg} \\
\text{else IDEXRegCompare} = 8
\]

Note that registers that can lead to data hazards range from 1-7 (register $0$ cannot lead to a data hazard), so a value “8” implies there is no data hazard at this stage.
Similarly, let MEMWBRegCompare be a 4-bit value where

\[
\text{if (EXMEMDataHazard} == 1 \&\& \text{EXMEMDstReg} \neq 0) \text{EXMEMRegCompare} = \text{EXMEMDstReg} \\
\text{else EXMEMRegCompare} = 8
\]

Let IDSrcReg1 and IDSrcReg2 be 4-bit values that identify source registers for the instruction in the IF/ID register. Recall that some registers have one source register (addi and lw), while others have two source registers (Rtype, sw, and beq). If there is only one source register, IDSrcReg2 = 9, which indicates there is no second source register. Note that the register indices can be found from the machine instruction in IF/ID. We refer to the first register, second, and third register fields in the machine instruction by IFIDField1, IFIDField2, and IFIDField3.

\[
\text{if (IFIDValid} == 0) \{\text{IDSrcReg1} = 9; \text{IDSrcReg2} = 9;\}
\]

\[
\text{else } \{
\text{if (IFIDOpcode} == \text{Rtype or SW or BEQ}) \{
\text{IDSrcReg1} = \text{IFIDRegField1}; \\
\text{IDSrcReg2} = \text{IFIDField2}; \\
\}
\text{else if (IFOpcode} == \text{LW or ADDI}) \{
\text{IDSrcReg1} = \text{IFIDRegField1}; \\
\text{IDSrcReg2} = 9; \\
\}
\text{else } \\
\text{IDSrcReg1} = 9; \\
\text{IDSrcReg2} = 9;
\}
\]

The second level of processing is to compare these register indices:

\[
\text{if ((IDSrcReg1} == \text{IDEXRegCompare}) \\
\text{|| (IDSrcReg2} == \text{IDEXRegCompare}) \\
\text{|| (IDSrcReg1} == \text{EXMEMRegCompare}) \\
\text{|| (IDSrcReg2} == \text{EXMEMRegCompare})) \{ \text{Stall PC and insert bubble; }\}
\]

\[
\text{else } \\
\text{PC} = \text{PC}+2; \\
\text{insert instruction into IDEX;}
\]

Appendix C shows the pipelined processor executing a program that multiplies 3 and 5.

Your task is to implement this pipelined PMIPSL. To modify your design from Part 1, consider the following:

- You must modify your pipeline registers so it now has IFIDValid, IDEXDataHazard, EXMEMDataHazard, IDEXDstReg, and EXMEMDstReg.

- Your controller in Part 1 was a sequential circuit with four states, which can be stored in a 2-bit register. But now your controller in Part 2 will be a combinational circuit. This means it has no clock input. On the other
hand, it must have additional inputs for the following registers: IFIDValid, IDEXDataHazard, EXMEMDataHazard, IDEXDstReg, EXMEMDstReg, and the opcode and register fields from the instruction stored in IF/ID.

- Note that the controller and pipeline registers will respond to inputs Reset and PCSrc. Reset is the highest priority input, then PCSrc.
  - Reset: When this is asserted, PC = 0, IFIDValid = 0, IDEXDataHazard = 0, EXMEMDataHazard = 0, IDEXBranch = 0, and EXMEMBranch = 0. You may have to clear other registers too.
  - PCSrc: You have to modify your pipeline registers so you can clear IDEX and EXMEM, as well as set IFIDValid to 0.

For Subproject 4, redesign your pipeline processor to be able to execute the program in IM1.V, which is the multiply program in Appendix C.

I will have the testbench and output trace available soon. It’ll probably look like the testbench for Subproject 1.
Appendix A

**MIPS-L Description:** Its data and address buses are 16-bits wide rather than 8. All instructions and operands (except ASCII characters) are 16-bits, i.e., “words” are 16 bits. Memory is byte addressable and is organized as BigEndian. Note that memory addresses of words are divisible by two.

**General Purpose Registers**

<table>
<thead>
<tr>
<th>Name</th>
<th>Reg #</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>1-2</td>
<td>values for results and expression evaluation</td>
<td>No</td>
</tr>
<tr>
<td>$t0-$t2</td>
<td>3-5</td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$sp</td>
<td>6</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra</td>
<td>7</td>
<td>return address</td>
<td>No</td>
</tr>
</tbody>
</table>

**Instruction Formats**

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Size</td>
<td>3 bits 3 bits 3 bits 3 bits 4 bits</td>
<td>ALL MIPS-L instructions 16 bits</td>
</tr>
<tr>
<td>R-format</td>
<td>op rs rt rd funct</td>
<td>Arithmetic instruction format</td>
</tr>
<tr>
<td>I-format</td>
<td>op rs rt Address/immediate</td>
<td>Transfer, branch, immediate format</td>
</tr>
<tr>
<td>J-format</td>
<td>op target address</td>
<td>Jump instruction format</td>
</tr>
</tbody>
</table>
Instructions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0 2 3 1 3</td>
<td>add $1,$2,$3</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0 2 3 1 2</td>
<td>sub $1,$2,$3</td>
</tr>
<tr>
<td>and</td>
<td>R</td>
<td>0 2 3 1 6</td>
<td>and $1,$2,$3</td>
</tr>
<tr>
<td>or</td>
<td>R</td>
<td>0 2 3 1 7</td>
<td>or $1,$2,$3</td>
</tr>
<tr>
<td>slt</td>
<td>R</td>
<td>0 2 3 1 4</td>
<td>slt $1,$2,$3</td>
</tr>
<tr>
<td>jr</td>
<td>R</td>
<td>0 7 0 0 5</td>
<td>jr $7</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>5 2 1 100</td>
<td>lw $1,100($2)</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>6 2 1 100</td>
<td>sw $1,100($2)</td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>2 1 2 (offset to 100)/2</td>
<td>beq $1,$2,100</td>
</tr>
<tr>
<td>addi</td>
<td>I</td>
<td>3 2 1 100</td>
<td>addi $1,$2,100</td>
</tr>
<tr>
<td>j</td>
<td>J</td>
<td>7 5000</td>
<td>j 10000</td>
</tr>
<tr>
<td>jal</td>
<td>J</td>
<td>1 5000</td>
<td>jal 10000 Loads return address in $7</td>
</tr>
</tbody>
</table>

Note that jal loads the return address into $7, and that $0 is always equal to zero.
Appendix B

Note that the PC logic has to be able to load the target branch address into the PC while the PC is in the stalled state. This can be done as follows. Take a look at Figure 1 and the circuits at the PC. Note that the PC input has a 2:1 multiplexer with select input PCSrc. For all instructions, except beq, PCSrc = 0 so that PC is incremented by 4. However, when the beq is being executed then PCSrc may equal 1 if the branch test is true. Then the branch target address is loaded from input 1 of the multiplexer.

We can modify this by inserting a new 2:1 multiplexer. The output of this multiplexer is attached to input 0 of the old multiplexer. The new multiplexer has its input 0 connected to the adder circuit that is equal to PC +4. Selecting this input would increment the PC. The new multiplexer’s input 1 is connected to the output of PC. Selecting this input would keep the PC value the same. The select input of this new multiplexer should be called “Stall”. “Stall” is connected to the controller. When the controller wants to stall the instruction fetch, it sets Stall to 1. Then the PC value stays the same. If the controller doesn’t want to stall the instruction fetch then it sets Stall to 0. Then the PC would normally be incremented by 4 unless there is a branch.
Appendix C

0: L0 addi $2,$0,3 // addi1
2: add $4,$0,$0
4: L1 beq $2,$0,L0 // beq1
6: addi $4,$4,5 // addi2
8: addi $2,$2,-1 // addi3
10: beq $0,$0,L1 // beq2
12: Next instruction // NI1 -- some random instruction
14: Next Instruction 2 // NI2 -- some random instruction
16: Next instruction 3 // NI3 --some random instruction, etc

<table>
<thead>
<tr>
<th>PC</th>
<th>I-Memory Output</th>
<th>IF/ID (src reg1, src reg2)</th>
<th>ID/EX [DataHaz, DstReg]</th>
<th>EX/MEM [DataHaz, DstReg]</th>
<th>MEM/WB</th>
<th>Comments: Actions by controller and datapath</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>addi1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>addi1 [0,]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>beq1</td>
<td>add (0,0)</td>
<td>addi1 [1,2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>addi2</td>
<td>beq1 (2,0)</td>
<td>add [1,4]</td>
<td>addi1 [1,2]</td>
<td></td>
<td>Stall: hazard via $2</td>
</tr>
<tr>
<td>6</td>
<td>addi2</td>
<td>beq1 (2,0)</td>
<td>Bubble</td>
<td>add [1,4]</td>
<td>addi1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>addi3</td>
<td>addi2 (4,)</td>
<td>beq1 [,]</td>
<td>Bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>beq2</td>
<td>addi3 (2,)</td>
<td>addi2 [1,4]</td>
<td>beq1 [,]</td>
<td>Bubble</td>
<td>Not branching</td>
</tr>
<tr>
<td>12</td>
<td>NI1</td>
<td>beq2 (0,0)</td>
<td>addi3 [1,2]</td>
<td>addi2 [1,4]</td>
<td>beq1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>NI2</td>
<td>NI1</td>
<td>beq2 [0,]</td>
<td>addi3 [1,2]</td>
<td>addi2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>NI3</td>
<td>NI2</td>
<td>NI1</td>
<td>beq2 [0,]</td>
<td>addi3</td>
<td>Branching to address 4, PCSrc = 1</td>
</tr>
<tr>
<td>4</td>
<td>beq1</td>
<td>Invalid</td>
<td>Bubble</td>
<td>Bubble</td>
<td>beq2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>addi2</td>
<td>beq1 (2,0)</td>
<td>Bubble</td>
<td>Bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>addi3</td>
<td>addi2 (4,)</td>
<td>beq1 [,]</td>
<td>Bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>beq2</td>
<td>addi3 (2,)</td>
<td>addi2 [1,4]</td>
<td>beq1 [,]</td>
<td>Bubble</td>
<td>Not branching</td>
</tr>
<tr>
<td>12</td>
<td>NI1</td>
<td>beq2 (0,0)</td>
<td>addi3 [1,2]</td>
<td>addi2 [1,4]</td>
<td>beq1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>NI2</td>
<td>NI1</td>
<td>beq2 [0,]</td>
<td>addi3 [1,2]</td>
<td>addi2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>NI3</td>
<td>NI2</td>
<td>NI1</td>
<td>beq2 [0,]</td>
<td>addi3</td>
<td>Branching to address 4, PCSrc = 1</td>
</tr>
<tr>
<td>4</td>
<td>beq1</td>
<td>Invalid</td>
<td>Bubble</td>
<td>Bubble</td>
<td>beq2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>addi2</td>
<td>beq1 (2,0)</td>
<td>Bubble</td>
<td>Bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>addi3</td>
<td>addi2 (4,)</td>
<td>beq1 [,]</td>
<td>Bubble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>beq2</td>
<td>addi3 (2,)</td>
<td>addi2 [1,4]</td>
<td>beq1 [,]</td>
<td>Bubble</td>
<td>Branching to address 0, PCSrc = 1</td>
</tr>
<tr>
<td>0</td>
<td>addi1</td>
<td>Invalid</td>
<td>Bubble</td>
<td>Bubble</td>
<td>beq1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>addi1 [0,]</td>
<td>Bubble</td>
<td>Bubble</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>