Laboratory Exercise 6
Pipelined Processors

Goals

After this laboratory exercise, you should understand the basic principles of how pipelining works, including the problems of data and branch hazards and possible remedies like forwarding and delayed branching. Finally, you should have an understanding of how the instructions are used to control different parts of a data path through a control unit.

Literature

- Patterson and Hennessy: Chapter 6.1–6.6, Appendix C.2
- MIPS Lab Environment Reference Manual

Preparation

Read the literature and this laboratory exercise in detail and solve the home assignments.

Home Assignment 1

The control unit in a MIPS processor can be described as a set of simple logical formulas. If you click on the Edit menu and then choose "Control unit..." an editor window opens, and you can write new control unit logic in a simple language based on JavaScript. Here is a part of the code used for the control unit logic in the simulator:

```
class Control
{
  in Instruction          // The entire instruction
  in BranchCondition:1    // From branch condition
  out BranchControl:2     // 0: beq, bne
                          // 1: j, jal
                          // 2: jr, jalr
                          // 3: default
  out RegWrite:1          // 0: Do not update register, 1: update register
  out RegDst:1            // 0: rt dest reg, 1: rd dest reg (default)
  out ALUSrc:1            // 0: R[rt] ALU src, 1: Imm. ALU src
  out NotJal:1            // 0: jal, jalr, 1: default
  out ALUOp:4             // 0: pass (default)
                          // 1: from func field
                          // 2: add
                          // 3: sub
                          // 4: and
                          // 5: or
                          // 6: xor
                          // 7: lui
                          // 8: shift
                          // 9: slt
  out MemRead:3           // 0: no read (default)
                          // 1: word
                          // 2: half word, signed
                          // 3: byte, signed
                          // 6: half word, unsigned
```

```
                        // 7: byte, unsigned
out MemWrite:2          // 0: No write (default)
                        // 1: word
                        // 2: half word
                        // 3: byte
out MemToReg:1          // 0: memory result to reg
                        // 1: ALU result to reg (default)
out UnknownOP:1

script

  // Declare variables with default values
  var VBranchControl;
  var VRegWrite;
  var VRegDst;
  var VALUSrc;
  var VNotJal;
  var VALUOp;
  var VMemRead;
  var VMemWrite;
  var VMemToReg;

  function ALU_Rformat(Func)
  {
    switch(Func){
    case 0x08:  // jr
      VBranchControl = 2;
      break;
    case 0x09:  // jalr
      VBranchControl = 2;
      VNotJal = 0;
      break;
    case 0x00:  // sll
    case 0x02:  // srl
    case 0x03:  // sra
    case 0x20:  // add
    case 0x22:  // sub
    case 0x24:  // and
    case 0x25:  // or
    case 0x26:  // xor
    case 0x27:  // nor
    case 0x2a:  // slt
      VRegWrite = ???;
      VALUOp = ???;
      break;
    default:  // unimplemented or unknown instruction
    }
  }

  function OnInstruction()
  {
    //  First compute good default values
    VBranchControl = 3;
    VRegWrite = 0;
```

```
VRegDst = 1;
VALUSrc = 0;
VNotJal = 1;
VALUOp = 0;
VMemRead = 0;
VMemWrite = 0;
VMemToReg = 1;
// The operator '>>>' means unsigned shift right
OpCode = Instruction.Get() >>> 26;
Func = Instruction.Get() & 0x3f;
// Compute specific instruction values
switch(OpCode) {
case 0x0:  // r-format
  ALU_Rformat(Func);
  break;
case 0x2:  // j
  VBranchControl = 1;
  break;
case 0x3:  // jal
  ???
  break;
case 0x4:  // beq
  if (BranchCondition.Get() == 1) VBranchControl = 0;
  break;
case 0x5:  // bne
  ???
  break;
case 0x8:  // addi
  VRegWrite = 1;
  VRegDst = 0;
  VALUSrc = 1;
  VALUOp = 2;  // add
  break;
case 0xa:  // slti
  ???
  break;
case 0xc:  // andi
  ???
  break;
case 0xd:  // ori
  ???
  break;
case 0xe:  // xori
  ???
  break;
case 0xf:  // lui
  ???
  break;
case 0x20:  // lb
  ???
  break;
case 0x21:  // lh
  ???
  break;
```

```
      case 0x23:  // lw
        ???
        break;
      case 0x24:  // lbu
        ???
        break;
      case 0x25:  // lhu
        ???
        break;
      case 0x28:  // sb
        ???
        break;
      case 0x29:  // sh
        ???
        break;
      case 0x2b:  // sw
        ???
        break;
      default:  // unknown op
        UnknownOP.Set(1);
      }
      // Set output values
      BranchControl.Set(VBranchControl);
      RegWrite.Set(VRegWrite);
      RegDst.Set(VRegDst);
      ALUSrc.Set(VALUSrc);
      ALUOp.Set(VALUOp);
      NotJal.Set(VNotJal);
      MemRead.Set(VMemRead);
      MemWrite.Set(VMemWrite);
      MemToReg.Set(VMemToReg);
    }
  end_script
  event Instruction OnInstruction()
}
```

It is only the code part between `script` and `end_script` that should be typed into the control unit editor window. The rest of the code is already known by the simulator.

The inputs are `Instruction` and `BranchCondition`. First, the code defines a set of variables for the outputs to be calculated. Then the function `OnInstruction` computes the outputs for the instruction, using the function `ALU_Rformat` as a helper for the register instructions. Finally, the variables are copied to the output signals.

Replace all "`???`" with correct code so that you produce a working control unit logic. The language is based on JavaScript, and you can use all standard Java constructs. The code above already contains all types of expressions needed, though.

Instruction Classes

Different classes of instructions need to use different parts of the available components in the data path. It is common to group MIPS instructions in four classes: *arithmetic*, *load*, *store*, and *branch* instructions. The instructions within one such class are quite similar, and it is often enough to understand one of them in order to understand all of them.

Assignment 1

The arithmetic instructions are sometimes also called *register instructions* because they perform an operation with two source registers and one destination register. You will now study how an arithmetic instruction goes through the pipeline.

Create a new project, type in the following program and build it. To be able to run it on the pipeline simulator open the program called *mipspipe2000.exe*. Choose Load Pipeline from the File menu, open the directory called S-script and choose the file called *s.dit* to load the small version of the pipeline. To open your program choose Open from the File menu or from the toolbar, go to the directory where you saved program, open the directory called *Objects* and the file of *.out* type.

```
        # Laboratory Exercise 7, Assignment 1
        # Written by Jan Eric Larsson, 26 November 1998

#include <iregdef.h>

        .set noreorder
        .text
        .globl start
        .ent start

start:  add     t0, t1, t2
        nop
        nop
        nop
        nop

        .end start
```

Insert some distinct values in t1 and t2 and start to execute the program by single stepping through each pipeline stage.

- What happens when the instruction goes through the first pipeline stage, the IF stage? Describe all signals, register changes, and other effects in detail.
- What happens in the second (ID) stage?
- What happens in the third (EX) stage?
- What happens in the fourth (MEM) stage?
- What happens in the fifth (WB) stage?
- What do IF, ID, EX, MEM and WB mean?
- How many clock cycles does it take for the result of the operation to be available in the destination register?
- In which pipeline stages do different arithmetic instructions differ?
- One stage is not used by arithmetic instructions. Which one? Why?

Assignment 2

Replace the instruction `add t0, t1, t2` in the program above with the instruction `lw t0, 0(t1)`, build, upload and investigate the program. Note that you must add a data variable from which to load a value.

- What happens in the different pipeline stages?
- What arithmetic operation does the ALU perform? Why?
- How many clock cycles does it take for the destination register to receive its value?
- Are all pipeline stages used? Explain.

Assignment 3

Now investigate the instruction `sw t0, 4(t1)` in the same way as with the other instructions above.

- What happens in the different pipeline stages?
- What arithmetic operation does the ALU perform? Why?
- Are all pipeline stages used? Explain.

Assignment 4

Finally, investigate the instruction `beq t0, t1, Dest` in the same way as with the other instructions above. Note that you must add a label named `Dest` somewhere.

- What happens in the different pipeline stages?
- What arithmetic operation does the ALU perform? Why?
- Are all pipeline stages used? Explain.

A Small Program Example

You will now investigate how a small program with several instructions goes through the pipeline so that the different stages of the instructions are executed in parallel. Study the following program, from Laboratory Exercise 1.

```
        # Laboratory Exercise 1, Home Assignment 2
        # Written by Jan Eric Larsson, 27 October 1998

#include <iregdef.h>

        .set noreorder      # Avoid reordering instructions
        .text               # Start generating instructions
        .globl start        # The label should be globally known
        .ent start          # The label marks an entry point

start:  lui     $9, 0xbf90  # Load upper half of port address
                            # Lower half is filled with zeros

repeat: lbu     $8, 0x0($9) # Read from the input port
        nop                 # Needed after load
        sb      $8, 0x0($9) # Write to the output port
        b       repeat      # Repeat the read and write cycle
        nop                 # Needed after branch
        li      $8, 0       # Clear the register

        .end start          # Marks the end of the program
```

Assignment 5

Upload the program above to the pipeline simulator and execute it step by step. Carefully note when the instructions are launched and when their results are ready.

Problems with Pipelining

Pipelining can make processors run up to *N* times faster than when they are executed one at a time, where *N* is the number of pipeline stages. However, there are several effects that cause problems and make it impossible to reach this efficiency in practice. One is that not all instructions use all pipeline stages. You will now investigate a few others.

Assignment 6

Run the following program on the pipeline simulator. Assign distinct values to t0, t1 and t3, and single step through the instructions.

```
        # Laboratory Exercise 7, Assignment 6
        # Written by Jan Eric Larsson, 26 November 1998

#include <iregdef.h>

        .set noreorder
        .text
        .globl start
        .ent start

start:  add     t2, t0, t1
        add     t4, t2, t3
        nop
        nop
        nop

        .end start
```

- After how many clock cycles will the destination register of the first add instruction, t2, receive the correct result value?
- After how many clock cycles is the value of t2 needed in the second instruction?
- What is the problem here? What is this kind of hazard called?

This kind of problem can be solved by code reordering, introduction of `nop` instructions, stalling the pipeline (hazard detection) and by forwarding. Explain when the first three methods can be used and how they work.

Both the hardware MIPS processor and the simulator uses *forwarding* to solve problems with data hazards. So far, you have used a version of the pipeline, S-script, that does not have forwarding. Switch to the pipeline in the directory *Xl-script*. This version does have forwarding. Then single step through the program above and study how the forwarding works. How does the forwarding unit detect that forwarding should be used? From where to where is data forwarded in the case above?

Assignment 7

Run the following program on the pipeline simulator. Use the simple version without forwarding (S-script). Assign the same value to t0 as to t1 and single step through the instructions.

```
        # Laboratory Exercise 7, Assignment 7
        # Written by Jan Eric Larsson, 22 February 1999

#include <iregdef.h>

        .set noreorder
        .text
        .globl start
        .ent start

start:  nop
        nop
        beq     t0, t1, start
        addi    t0, t0, 1
        nop
        nop
```

```
        nop

        .end start
```

- How many cycles does it take until the branch instruction is ready to jump? What has happened with the following `addi` instruction while the branch is calculated?
- What is the problem here? What is this kind of hazard called?
- What are the possible solutions to this problem?
- Switch to the forwarding version (Xl-script) again. How does this version handle `beq`?

Assignment 8

Run the following program on the pipeline simulator. Assign distinct values to t0 and t1 and set t2 to contain the address of a memory location where you know the contents. Then single step through the instructions.

```
        # Laboratory Exercise 7, Assignment 8
        # Written by Jan Eric Larsson, 22 February 1999

#include <iregdef.h>

        .set noreorder
        .text
        .globl start
        .ent start

start:  lw      t0, 0(t2)
        add     t1, t1, t0
        nop
        nop
        nop

        .end start
```

- After how many clock cycles will the destination register of the load instruction, t0, receive the correct result value?
- After how many clock cycles is the value of t0 needed in the `add` instruction?
- What is the problem here? What is this kind of hazard called?

This kind of problem can be solved with forwarding or hazard detection and stalling, just as other data hazards, but most MIPS implementations do not have these for load.

- What are the alternative solutions that can be used?
- Does the forwarding version of the simulator, Xl-script, handle the problem with delayed load?

Assignment 9

The control unit in a MIPS processor can be described as a set of simple logical formulas. If you click on the Edit menu and then choose "Control unit..." an editor window opens, and you can write new control unit logic in a simple language based on JavaScript. Enter the control unit logic code from Home Assignment 1 in the control unit editor window. Run the simulator and verify that your control unit works correctly by running some of the previous assignments again.

Assignment 10

Study the following code. It consists of a loop in which the register t2 works as a loop counter and is counted down to zero. Each time the loop is executed, one word of memory is loaded, incremented by 1 and stored again, and then the same is done with another word.

```
        # Laboratory Exercise 7, Assignment 10
        # Written by Jan Eric Larsson, 22 February 1999

#include <iregdef.h>

        li      t2, 1000
Loop:   lw      t0, 0(s0)
        nop
        add     t0, t0, s1
        sw      t0, 0(s0)
        lw      t1, 4(s0)
        nop
        add     t1, t1, s1
        sw      t1, 4(s0)
        subi    t2, t2, 1
        bne     t2, zero, Loop
        nop
```

Write a full program with this loop in it, test it on the ordinary simulator and then on the lab processor.

The two functions `timer_start()` and `timer_stop()` can be used to measure execution time. The function `timer_start()` tells the timing to start, while `timer_stop()` stops the timer and returns the time since the start, in microseconds on the lab computer and in clock cycles on the simulator.

Use these two functions to time the loop above. Choose a initial counter value for t2 so that the loop takes several milliseconds. Write down how many milliseconds it takes to execute the loop.

Assignment 11

The loop shown in Assignment 10 can be written more efficiently. How? Try to reorder to code in the loop so that it becomes more efficient and time the new version of the loop. How much faster can you make it?

Conclusions

Before you finish the laboratory exercise, think about the questions below:

- How can a pipelined processor be faster than one without pipelining?
- What are the special problems that appear in pipelining?
- How can these problems be solved?
- Is the lab computer more like S-script or Xl-script?
- Is there a difference in writing compilers for pipelined processors?
- Which is faster: straight code or code with many branches?
- What does RISC mean? What does CISC mean?
- Is the Pentium processor pipelined? Pentium Pro? Pentium 2?