Project 3: Pipelined RiSC-16 (10%)

ENEE 350: Computer Organization, Fall 2009
Assigned: Thursday, Oct 29; Due: Tuesday, Nov 10

Purpose

This project is intended to help you understand in detail how a pipelined microprocessor works. You will build a pipelined RiSC-16, complete with *data forwarding*, simple *branch prediction*, and *speculative execution*. The next project will add *caches* and *precise interrupts*. For details on the RiSC-16 pipeline, see the document *The Pipelined RiSC-16* on the class website.

Pipelines

In the previous project, you built a sequential processor, similar to what is described in the document *RiSC-16: Sequential Implementation*. The document shows the control flow and data flow for each instruction, as well as the final hardware implementation that changes its dataflow based on the instruction opcode. In a sequential implementation, the entire instruction must be executed before the next clock, at which point the results of the instruction are latched in the register file or data memory. This results in a relatively long clock period.

The computer market is not fond of slow clocks, however. Increased clock speeds are possible as the amount of logic between successive latches is decreased. If execution is sliced up into smaller sub-tasks, the clock can run as fast as the longest sub-task. Theoretically, a pipeline of N stages should run with a clock that is N times faster than a sequential implementation. For many reasons, this theoretical limit is never reached, due to latch overhead, sub-tasks of unequal length, etc. Nonetheless, extremely fast clock rates are possible. Slicing up the instruction execution this way is called *pipelining*, and it is exploited to great degree in nearly every aspect of modern computer design, from the processor core to the DRAM subsystem, to the overlapping of transactions on memory and I/O buses, etc.

The RiSC-16 pipeline is shown in Fig. 1 on the next page. It is similar to the 5-stage DLX/MIPS pipeline that is described in both *Hennessy & Patterson* and *Patterson & Hennessy*, and it fixes a few minor oversights, such as lack of forwarding to store data, lack of forwarding to comparison logic in decode implementing the 1-instruction delay slot, etc. This pipeline adds in forwarding for store data and eliminates branch delay slots. As in the DLX/MIPS, branches are predicted not taken, though implementations of more sophisticated branch prediction are certainly possible.

In the figure, shaded boxes represent clocked registers; thick lines represent 16-bit buses; thin lines represent smaller data paths; and dotted lines represent control paths. The figure illustrates how pipelining is achieved: the sub-tasks into which instruction execution has been divided are instruction fetch, instruction decode, instruction execute, memory access, and register-file writeback. Each of these sub-tasks, which is executed by dedicated hardware called a pipeline stage, produces intermediate results that must be stored before an instruction may move on to the next stage. By breaking up execution into smaller sub-tasks, it is possible to overlap the different sub-tasks of several different instructions simultaneously. If the intermediate results of the various sub-tasks are not stored, they would be lost: during the next cycle another instruction would use the same hardware for its own task. For instance, after an instruction is fetched, it is necessary to store the fetched instruction somewhere, because the output of the instruction memory will be different on the following cycle—the fetch stage will be fetching a completely different instruction.

The storage locations for the intermediate results are called *pipeline registers*, and the figure illustrates their contents. It is common to label a pipeline register with the two stages that it divides. For example,

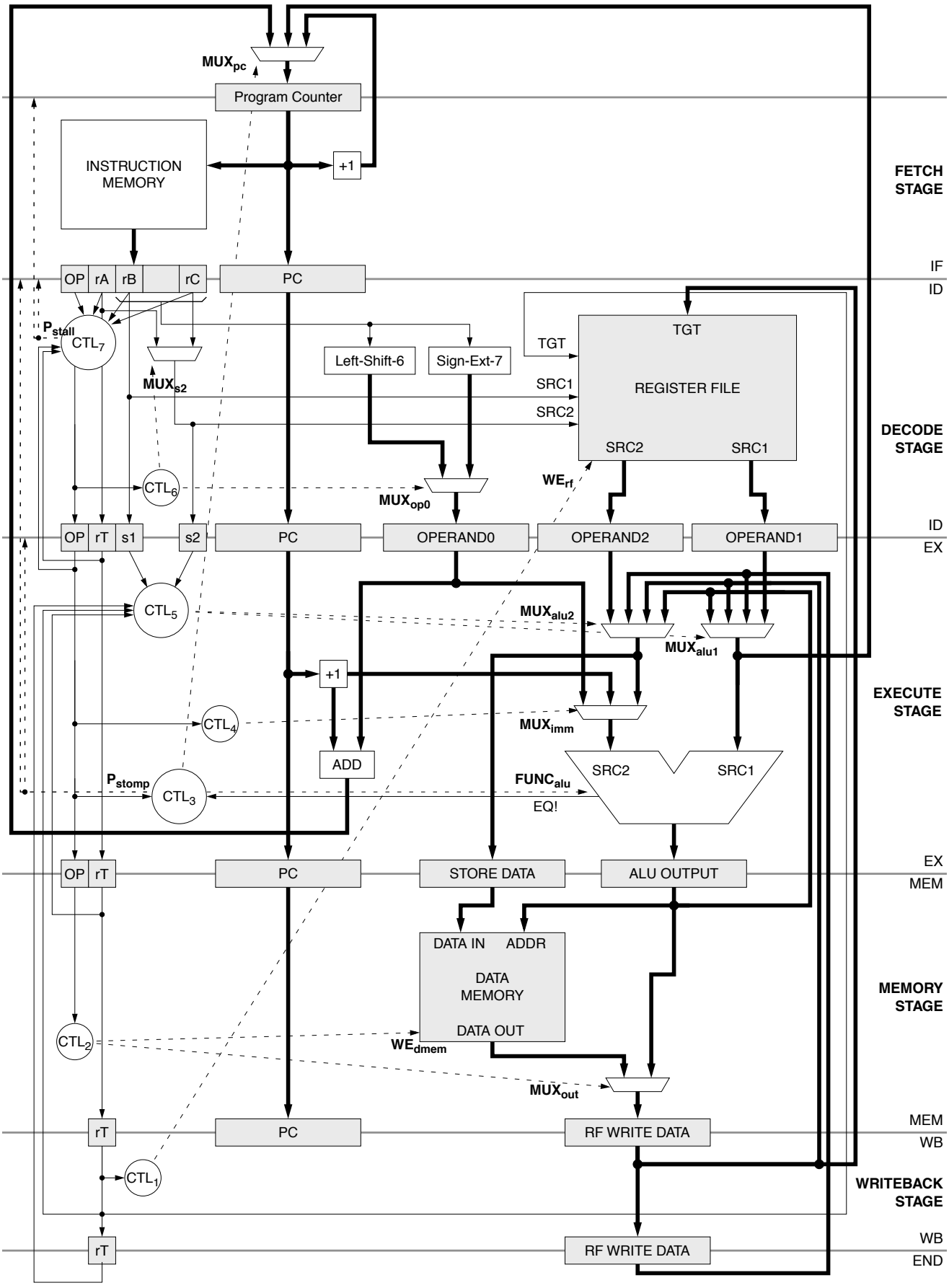

Fig. 1: RiSC-16 5-stage pipeline

the pipeline register that divides the instruction fetch (IF) and instruction decode (ID) stages is called the *IF/ID register*; the register that divides the instruction execute (EX) and memory-access (MEM) stages is called the *EX/MEM register*; etc.

Note that neither the WB/END register nor the data-forwarding path it supports is present in the DLX/MIPS architecture described by Hennessy & Patterson. The DLX/MIPS assumes a half-cycle register-file access, so that the writeback stage completes in the first half of the cycle and the register-file read component of the decode stage happens in the second half of the cycle. This allows data to be forwarded from the writeback stage to the decode stage directly. Otherwise such forwarding is impossible, unless the register file has a pass-through design that connects data-in to data-out whenever reading and writing the same register. If the register file does not do such forwarding, then the data written to the register file is only available on the following cycle. Thus, there must be a path to forward data to the instruction in the decode stage at the same time as the instruction writing to the register file in the writeback stage. This is the function of the WB/END register and the forwarding stage it represents.

C Implementation

The descriptions given above are suitable for an actual circuit design, but we will not get that detailed. Instead, you will build this pipeline in C, which should be much easier. I will give you skeleton code (available on the class project website) that does much of the grunt-work for you.

Datapath

The shared-bus style of Homework 3's implementation is not suitable for pipelining, because it is impossible for more than one instruction to use the bus at the same time. So for this project we will use a datapath similar to the one described in Chapter 6 of *Patterson and Hennessy*. Of course, since the MIPS and RiSC-16 are slightly different, we will have to make a few minor changes to the book's datapath.

1. Instead of a "4" input in the PC's adder, we will use a "1", since the RiSC-16 is word-addressed instead of byte-addressed.
2. The instruction bit fields have to be modified to suit the RiSC-16's instruction-set architecture.
3. The "shift left 2" component is not necessary, since the immediate values for branches and the PC use word-addressing.

One of the most noticeable differences between Project 3 and the pipelining done in the book is that we add a pipeline register AFTER the write-back stage (the **WBEND** pipeline register). This will be used to simplify data forwarding so that the register file does not have to do any internal forwarding.

Memory

Note in the typedef of **state_t** below that there are two memories: **instrMem** and **dataMem**. When the program starts, read the machine-code file into BOTH **instrMem** and **dataMem** (i.e. they'll have the same contents in the beginning). During execution, read instructions from instrMem and perform load/stores using **dataMem**. That is, **instrMem** will never change after the program starts, but **dataMem** will change.

C-Language Pipeline Registers

To simplify the project and make the output formats uniform, you must use the following structures *without modification* to hold pipeline register contents. Note that the entire instruction is passed down the pipeline.

```
#define MAXMEMORY 65536 /* maximum number of data words in memory */
#define NUMREGS 8 /* number of machine registers */

#define ADD 0
#define ADDI 1
#define NAND 2
#define LUI 3
#define LW 4
#define SW 5
#define BEQ 6
#define JALR 7

#define NOP_INSTRUCTION 0x0000

typedef struct IFIDStruct {
        short instr;
        short pcPlus1;
} IFID_t;

typedef struct IDEXStruct {
        short instr;
        short pcPlus1;
        short readReg1;
        short readReg2;
        short offset;
} IDEX_t;

typedef struct EXMEMStruct {
        short instr;
        short branchTarget;
        short aluResult;
        short readReg2;
} EXMEM_t;

typedef struct MEMWBStruct {
        short instr;
        short writeData;
} MEMWB_t;

typedef struct WBENDStruct {
        short instr;
        short writeData;
} WBEND_t;

typedef struct stateStruct {
        short pc;
        short instrMem[MAXMEMORY];
        short dataMem[MAXMEMORY];
        short reg[NUMREGS];
        short numMemory;
        IFID_t IFID;
        IDEX_t IDEX;
        EXMEM_t EXMEM;
        MEMWB_t MEMWB;
        WBEND_t WBEND;
        int cycles; /* number of cycles run so far */
} state_t;
```

Problem

Basic Structure

Your task is to write a cycle-accurate simulator for the RiSC-16. At the start of the program, initialize the pc and all registers to zero. Initialize the instruction field in all pipeline registers to the nop instruction (0xe000).

The main run() function will be a loop, where each iteration through the loop executes one cycle. At the beginning of the cycle, print the complete state of the machine (use the **printState** function at the end of this handout without modification). In the body of the loop, you will figure out what the new state of the machine (memory, registers, pipeline registers) will be at the end of the cycle. Conceptually all stages of the pipeline compute their new state simultaneously. Since statements execute sequentially in C rather than simultaneously, you will need two state variables: **state** and **new**. The variable **state** contains the state of the machine while the cycle is executing; the variable **new** will be the state of the

machine at the end of the cycle. Each stage of the pipeline will modify the **new** variable using the current values in the **state** variable. E.g. in the ID stage, you will have a statement like

```
new.IDEX.instr = state.IFID.instr;
```

(to transfer the instruction in the IFID register to the IDEX register)

In the body of loop, you will use **new** ONLY as the target of an assignment and you will use **state** ONLY as the source of an assignment (e.g. **new**... = **state**...). In general, **state** should never appear on the left-hand side of an assignment, and **new** should never appear on the right-hand side of an assignment.

Your simulator must be pipelined. This means that the work of carrying out an instruction should be done in different stages of the pipeline as done in the textbook, and the execution of multiple instructions should be overlapped. The WB stage should be the only stage that writes to the register file; other stages should write to the pipeline registers. The ID stage should be the only stage that reads the register file; the other stages must get the register values from a pipeline register.

Here's the main simulator loop:

```
while (1) {

        printState(state);

        /* check for halt */
        if (opcode(state.MEMWB.instr) == HALT) {
                printf("machine halted\n");
                printf("total of %d cycles executed\n", state.cycles);
                exit(0);
        }

        memcpy(&new, state, sizeof(state_t);
        new.cycles++;

        /* --------------------- IF stage --------------------- */

        /* --------------------- ID stage --------------------- */

        /* --------------------- EX stage --------------------- */

        /* --------------------- MEM stage --------------------- */

        /* --------------------- WB stage --------------------- */

        memcpy(state, new, sizeof (state_t));
                /* this is the last statement before end of the loop.
                It marks the end of the cycle and updates the current
                state with the values calculated in this cycle */
}
```

Please use some sort of commenting like this so that we can easily identify what stages or your simulated pipeline are doing what. Without commenting, we will assume that everything happens all at once (i.e. no pipelining, therefore wrong).

Halting

At what point does the pipelined computer know to halt? It is incorrect to halt as soon as a **halt** instruction is fetched because if an earlier branch was actually taken, then the machine might actually branch around the **halt** instruction, in which it will be squashed and not executed.

To solve this problem, halt the machine when a **halt** instruction reaches the MEMWB register. This ensures that previously executed instructions have completed, and it also ensures that the machine won't branch around this **halt**. This solution is shown above; note how the final **printState** call before the check for halt will print the final state of the machine.

Begin Your Implementation Assuming No Hazards

The easiest way to start is to first write your simulator so that it does not account for data or branch hazards. This will allow you to get started right away. Of course, the simulator will only be able to correctly run assembly-language programs that have no hazards. It is thus the responsibility of the assembly-language programmer to insert nop instructions so that there are no data or branch hazards. This means putting a number of nops in an assembly-language program after a branch and a number of nops in an assembly-language program before a dependent data operation (it is a good exercise to figure out the minimum number needed in each situation).

Finish Your Implementation by Accounting for Hazards

Modifying your first implementation to account for data and branch hazards will probably be the hardest part of this assignment.

Use data forwarding to resolve most data hazards. I.e. the ALU should be able to take its inputs from any pipeline register (instead of just the **IDEX** register). There is no need for forwarding within the register file (as the book has). For this case of forwarding, you'll instead forward data from the **WBEND** pipeline register. Remember to take the most recent data (e.g. data in the **EXMEM** register gets priority over data in the **MEMWB** register). **Only forward data to the EX stage**.

You will need to stall for one type of data hazard: a **lw** followed by an instruction that uses the register being loaded.

Static Branch Prediction

Assume branch-not-taken for forward branches (those with a *positive* immediate value) and branch-taken for backward branches (those with a *negative* immediate value). This requires you to discard instructions if it turns out that the predicted direction was incorrect. To discard instructions, change the relevant fields in the pipeline to the **nop** instruction (0x0000).

Running and Demonstrating Your Program

Your simulator should be run using the same command format specified in Project 1, that is:

```
simulate code > output
```

I will give you skeleton code to use as a starting point. You should use the solution assembler from Project 1 to create the machine-code file that your simulator will run (since that's how we'll test it).

Grading and Formatting

We will grade almost solely on functionality. In particular, we will run your program on various assembly-language programs and check the contents of your memory, registers, and pipeline registers at each cycle. Most of these assembly-language programs will have hazards; a few will be hazard-free.

Since we'll be grading on getting the exact right answers (both at the end of the run and being cycle-accurate throughout the run), it behooves you to spend a lot of time writing test assembly-language programs and testing your program. Programs that are not doing exactly what they're supposed to on every cycle will be penalized heavily, so check carefully.

We will use a program to compare your output against a solution output. So it's very important that you follow these exact formatting rules:

1. Don't modify **printState** at all.

2. There should be ONLY ONE call to **printState** in your program. Do not put in any extra **printState** calls (you can put these in for debugging, but take them out before submitting the program).
3. Make sure to initialize all values correctly.

   a. **state.numMemory** should be set to the number of words in the machine-code file.

   b. **state.cycles** should be set to 0.

   c. **pc** and all registers should be set to 0.

   d. the instruction field in all pipeline registers should be set to the **nop** instruction.
4. Check your program's output on the sample assembly-language programs and output that I will give you.

Having events happen on the right cycle will also be very important (e.g. stall the exact number of cycles needed, write the branch target into the PC at exactly the right cycle, halt at the exact right cycle, stalling only when needed).

Turning in the Project

I will get the submit function up and running shortly … there will be an autograder that returns the results of correctness tests to you within several minutes. You need submit nothing else.