B443 Computer Architecture, Spring 2018

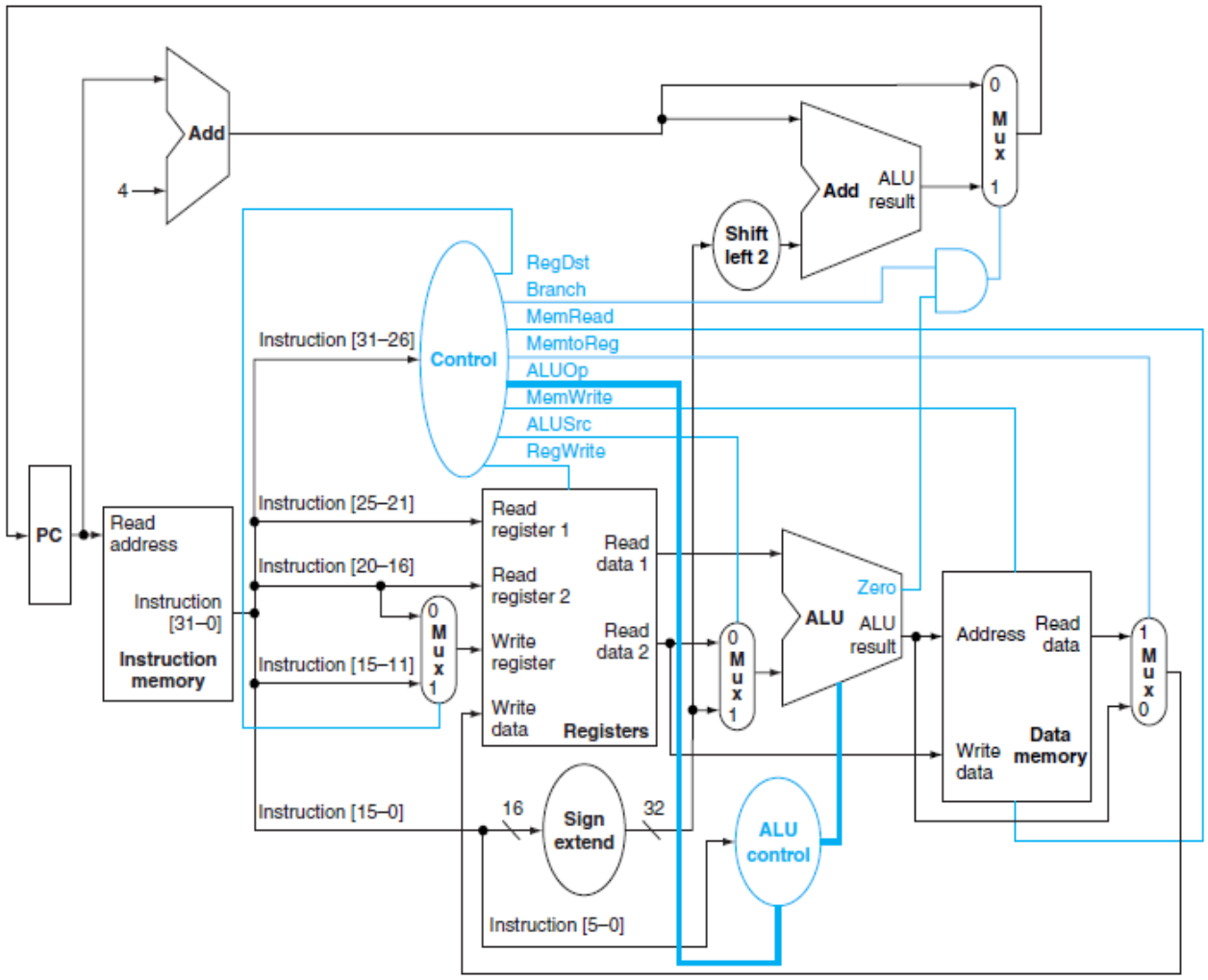
Project 1: Processor Simulator

Issued: Saturday, February 17, 2018

Due: Saturday, March 3, 2018

Processor Simulator

In this project you will implement a single cycle MIPS simulator in Java. The block diagram of the single cycle MIPS is shown below.



We have provided a partial solution with a graphical interface, the ability to load an assembled MIPS assembly file, and some other details not relevant to this course. Some sections of the code are marked "TODO." The implementation of these sections will be left to you. The incomplete files will be listed at the end of the project description.

Compilation and Execution

To compile the project, enter the "src" directory, and run "javac -d .. procsimu/*.java". To run it, from the project root (the directory containing "src"), run "java procsimu.Control".

Some assembled test programs have been provided in the "asm" directory. Their original MIPS source can be found in the "asm/src" directory.

Note: If you wish to write your own test programs, they can be assembled with the Mars simulator; however, Mars produces assembler output in little-endian format, while our simulator is designed to execute big-endian machine code. A Python script has been provided that will flip the endianness of an assembled MIPS file. You can run it with "python flip.py [INPUT].mips".

Usage

The messages related to the execution of each one of the five MIPS stages are displayed in the boxes Fetch, Decode, Execute, Memory and Write Back. The application terminates when you click "Quit."

The application should simulate the functionality of the five processor stages. The stages should work in sequence, within one clock cycle. For example, slides 10 to 14, Lecture "W06-M Pipeline" explain the sequence of steps for instruction "load word (lw):"

1. fetch,
2. decode,
3. execute,
4. memory,
5. write back.

The executable simulator application is provided in the file "Project_1.jar." Once you provide the missing code into the partial source code (provided), your simulator should work like (or better than!) "Project_1.jar." The partial solution provides the following classes.

- **Control:** The main class for the simulator which models the CPU control unit. It contains all other objects and controls the events between them.
- **gui.Output:** Contains the components involving user interaction with the simulator.
- **Stage**: Abstract class for each of the five pipeline stages.
- **StageIF, StageID, StageEXE, StageMEM, StageWB**: Handle the respective stages.
- **Configuration:** Provides configuration options, such as the size of the memory.

- **comp.Const:** Defines constants, such as instruction opcodes.
- **comp.Conversion:** Provides utility functions for extracting register numbers, immediates, etc. from encoded instructions.
- **comp.Memory**: Implements hardware memory as either a register file or main memory.
- **comp.Latch**: Models a hardware latch that only reflects a new value written to it on a clock change.
- **comp.Register**: Models a register.
- **comp.ClockChangeListener**: Interface implemented by Latch, for example; allows the Control class to notify other components of a clock cycle.
- **comp.WorkListener**: Interface implemented by each stage which allows the Control class to trigger execution of the stage.
- **util module:** Provides utility functions for parsing arguments passed to the simulator.

You are supposed to simulate the execution of the following instructions. Some of them, like AND, have been implemented for you as an example.

R-type instructions:

ADD, SUB AND, OR, XOR, SLL, SRL, SLT

I-type instructions:

ADDI, ANDI, ORI, XORI, SUBI

Load/store word instructions:

LW, SW

Branch instructions:

BEQ, BNE

The files with sections marked “TODO” are:

- Control.java
- comp/Const.java
- StageIF.java
- StageID.java
- StageMEM.java
- StageEXE.java
- StageWB.java

Hint: The >> operator in Java performs an arithmetic shift. SRL is a logical shift, which can be simulated with the >>> operator.