Building a Pipelined Computer in the Architecture Laboratory

Richard J. Reid
Michigan State University

Abstract - This computer architecture laboratory uses an object-oriented approach to provide a simulation modeling language. This language allows students to complete models of *real* pipelined computers. The modeling language is implemented as a class library for C++. Using this library, students are able to complete working models of an actual Silicon Graphics microprocessor, the MIPS 4000. Student work is easily validated since a correct model allows simulated execution of the code produced for a standard model of the microprocessor.

Introduction

Students come to this Computer Architecture course with a two-semester background in C++ programming, and one semester of introductory machine organization and assemble-language programming. The latter course, using the text by Maccabe [1], includes four laboratory sessions in which students use the digital simulator described below to implement combinational gating networks and simple sequential machines.

The laboratory activity reported here supports this Computer Architecture course and is a required course for Computer Science and Computer Engineering majors, and is elected by many Electrical Engineering and other students. The course is completed by 60 students each semester. This course uses the text by Patterson and Hennessy [2] with supplementary material from the MIPS Microprocessor User's Manual [3]. Although each student is completing a functionally equivalent microprocessor, each is assigned personalized data-flow pathways for which their component interconnections and control must be customized.

The Pipelined Computer

To accommodate the design and implementation of complex digital networks and computing structures, laboratories are turning to simulation, [4,5]. Simulation allows the convenient modeling of extensive designs.

In keeping with the architecture of all modem computers, those implemented in this laboratory use pipelining for efficient execution of instructions. The figure shows the three stages, fetch, decode and execute, that are implemented.

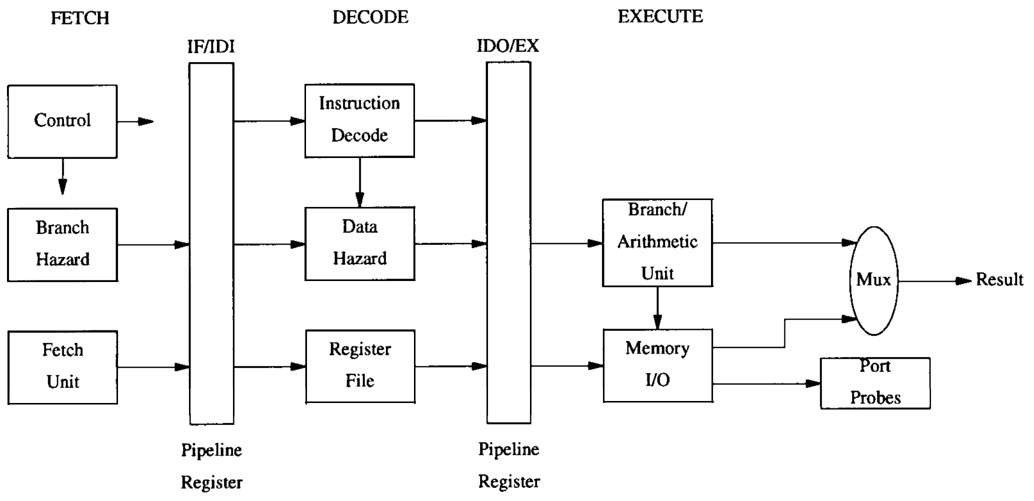

Figure 1. The Pipelined Computer

As the system clock ticks in this model, instructions pass from left to right, and partial completions are accomplished within each of the segments, in assembly-line fashion.

The Essential features of pipelining encountered in this design include:

- Separation of instruction execution into multiple stages.
- Simultaneous processing of different instructions within the multiple stages.
- Detection of data and branch dependencies.
- Stalling the pipeline when necessary to prevent hazardous execution.

Simulation Modeling

Object-oriented languages such as C++ allow class definitions which eliminate the need for using special-purpose simulators in many cases. Simulating digital logic components and computer architectures is one case where the simulation model can be effectively and conveniently expressed in the programming language itself.

A class library supporting: schematic organization, multi-level digital-signal representation, and implementations of a modest set of component primitives has been developed. This library supports two forms of hierarchical arrangements: first, the digital signals themselves can be expressed as vectors (as for a bus), either directly or by composition, and second, the digital components can be arranged hierarchically as modules, and the modules can be used in an identical manner in which the primitives are used. This library code described here is free software and is available from the author.

The animation features provided with the simulation model allow seeing any collection of logical signals as their values sequence over time. The model displays the system clock ticking, and the instructions progressing through the pipeline can be viewed. While the simulation would normally advance too rapidly to allow viewing a meaningful animation, interactive control of the simulation time is provided, so every change *can* be observed.

Networks to be simulated are implemented by declaring the signals involved and connecting those signals to the component inputs and outputs. The general format of component specification is:

```
<Component Type> (<Schematic Position>, <Input(s)>, <Output(s)>);
```

as C++ function invocations.

Consider the following simple example, written in C++, of a two-input And gate activated by Switches and monitored by a digital Probe:

```
#include <sim.h>

main()
{
 Signal a, b, c; // Declaration of signals

 Switch (" la", a, 'a' );// Position Switch at schematic position "la", and associate keyboard 'a' key with Signal a

 Switch (" la", b, 'b' );// Associate keyboard 'b' key with Signal b

 And ( "lb", ( a, b), c );// Group scalar signals as an input vector

 Probe ( " lc", c ); // Display the output

 sim(); //Begin simulation
}
```

Since C++ is strongly typed, all Signals must be declared before usage. The Signals a, b, c appear as scalars in this case; however, Signal declaration is for signal vectors, and the default form here creates signal vectors of unit length.

The <Schematic Position> entries position the components on a two-dimensional grid in the simulation-time display-window. The simulation is interactive, and the Switch components provide keyboard links to signals so the logical signals may be toggled during simulation time.

Components may generally have multiple inputs and outputs. The second argument to the And above is *composed* using the overloaded comma operator. The inner parentheses escape from the outer level that is doing argument-list formation, to the level where comma is just another C++ operator--overloaded for class Signal to allow composition in this case. The simulation-time display is shown in Figure 2.

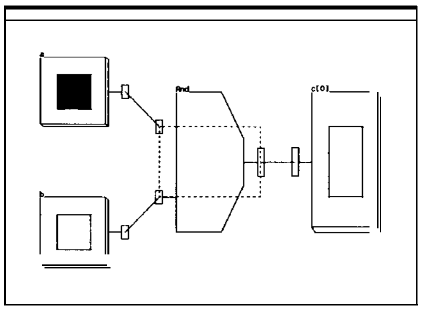

Figure 2. And Network

Here the keyboard keys 'a' and 'b' have been manipulated (in producing the screen-dump Figure) to provide logical one and zero And gate activations respectively. The Probe shows the gate output as logical zero.

While only a simple And gate is shown above, the And and other components allow a simple expression of more complicated gate arrays. These alternative forms are described by the following:

```
void And ( const SD & sd, const Signals & in, const Signals & out );
//This is a simple single And when the "out" is length one.
//It's a vector of And's if "out" is a vector. In this case,
//in.length = N * out.length, and each And has N inputs.
```

Simple gates, such as the And above, and more complicated primitive types, such as an arithmetic logic unit (Alu), are used to construct the several modules shown in Figure 1, throughout the term. Shown below is a typical module, the Branch/Arithmetic Unit.

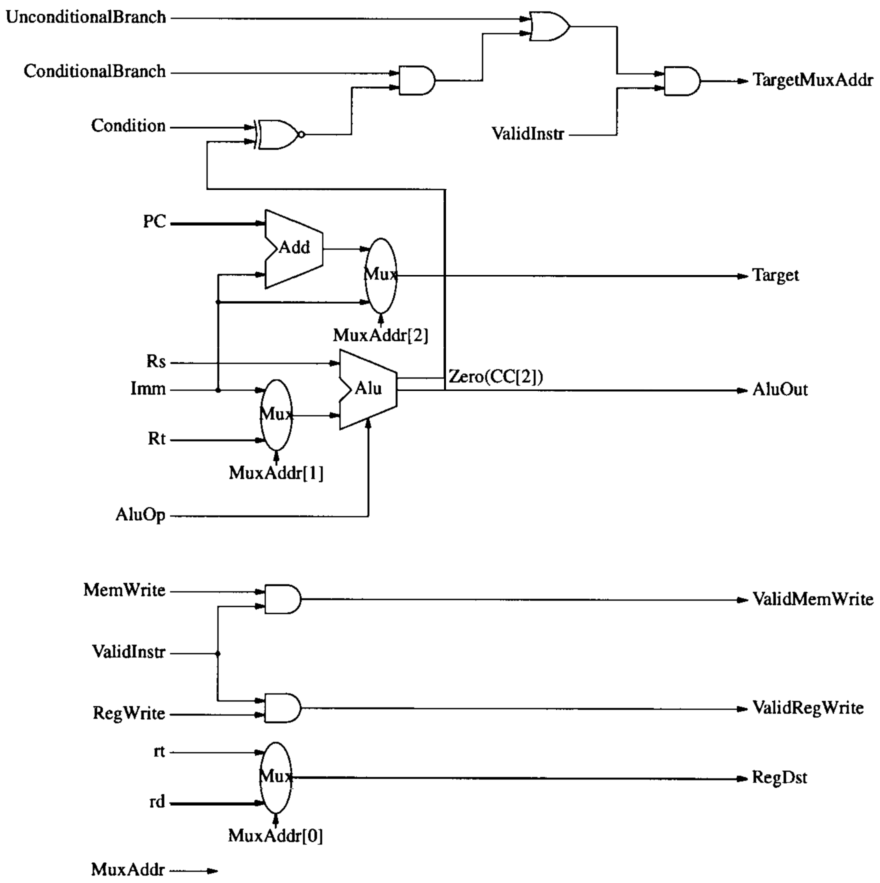

Figure 3. Branch/Arithmetic Unit

Testing of these individual modules, as they are completed, helps insure they will function properly, when they are finally interconnected in the pipeline machine, during the last few weeks of the semester. The culmination of building a model of a standard processor is running *actual* code produced for a ***real*** machine. We see below a typical testing program in its three forms: 1) The hex codes of execution, 2) The assembly-language version, and 3) The C-language version. This section of code tests that values can be written to a region of Ram and subsequently retrieved and checked.

```
Hexadecimal            Assembly-Language        C-Language

2004AAAA               addi $4, $0,0xAAA4       # register int * ram= (int *) OxAAAA;
20050000               addi $5, $0,0            # register int i = O;   I* scaled *I
20060000               addi $6, $0,0            # register count = 0x8000;
2007EEEE               addi $7, $0,0xEEEE       # register int * stderr = (int *) OXEEEE;
20080040               addi $8,$0,64            # loop limit  /* scaled */
00002820    start:     add $5,$0,$0             # for(i=0;i<16; i++)
10A8OOO5              A:beq $5, $8, .endforl
00A65020               add $10,$5,$6            # *(ram+i) = i + count;
00855820               add $11,$4,$5
AD6A0000               Sw   $10,0($11)
08000006               j A
20A50004               addi $5, $5,4            # /* i++: scaled */
00002820    .endfor1:  add $5,$0,$0             # for(i=0;i<16; i++)
10A80008              B:beq $5,$8, .endfor2
00A65020               add $10,$5,$6            # if (*(ram+i) == (i + count) )
00855820               add $11,$4,$5
8D6C0000               lw   $12,0($11)
154C0002               bne $10,$12, .else
20AD0030               addi $13,$5,0x30         #  *stderr = i + 0x30;
ACED0000               Sw   $13,0($7)
0800000D    else:      j B
20A50004               addi $5,$5,4             # /* i++: scaled */
0800000C    .endfor2:  j endfor1                # goto lastfor;
00C63020               add $6,$6,$6             # count *= 2;
```

Conclusions

By using a simulation model of areal pipelined machine, in the computer architecture course laboratory, students complete a realistic design. Student work is easily validated in this laboratory since a correct model allows execution of the standard codeforthe MIPS microprocessor. Opportunities exist for reasonable extensions in this laboratory work. The microprocessor currently implemented has only thee stages of pipelining--this can be elaborated to the four or more commonly used. Cache implementations of the instruction memory is planned as an addition for these projects in the future.

References

1. B. Maccabe "Computer Systems: Architecture, Organization, andProgramming," RichardD. Irwin, Inc., Homewood,IL, 1993.
2. D. Patterson and J. Hennessy, "Computer Organization&Design,TheHardware/Software Interface," Morgan Kaufmann Publishers, San Mateo, California, 1994.
3. J. Heinrich, "MIPS R4000 User's Manual," PTR Prentice Hall, Englewood Cliffs, New Jersey, 1993.
4. M. Singh, "Role of Circuit and Logic Simulation in the EE Curriculum," *IEEE Trans. Educ., Vol. E-32, No. 43, pp. 411-414, August, 1989.*
5. F. Tangorra, "The Role of the Computer Architecture Simulator in the Laboratory," *ACM SIGCSE Bulletin,* Vol. 22, No. 1, pp. 5-10, June, 1990.

RICHARD J. REID (reid@cps.msu.edu) received B.S. and M.S. degrees in electrical engineering from Iowa State University in 1955 and 1956, respectively, and the Ph.D. degree in electrical engineering from Michigan State University in 1959. Since 1965 he has been professor of Computer Science at Michigan State University. His current interests are in computer architecture and simulation.