EE 469 Lab 5

Designing a Pipelined CPU

Bring it Together Phase 2….

University of Washington - Department of Electrical Engineering

Matt Staniszewski, Hoon Kwon, Andrew Lawrence, and James K. Peckol

---

Introduction:

In this fifth and final phase of the project to design and implement a CPU, we will begin with the single cycle architecture that we developed previously as a base. We will now convert and extend the single cycle datapath into a pipelined datapath and add an appropriately modified control unit and control scheme to manage dataflow through the system. We will continue to work with the same instruction set that we used for the single cycle design.

In support of the pipelined datapath, the CPU must manage data and control/instruction hazards. To this end, the CPU will support data forwarding as well as accommodate the load, branch, and jump instructions. We will not address structural or cache hazards.

As in the previous phases of this project, we will utilize our knowledge of the C language and the MIPS instruction set by working with a simple program, hand compiling it then loading into memory and executing it on our pipelined CPU. Execution of the program will follow the *fetch, decode, execute, write back, next* instruction cycle in a pipelined context.

As a final step in the project, we will compare the performance of the single clock cycle design with the pipelined version.

Prerequisites:

Familiarity with the Quartus development environment and the signal tap logic analyzer. A continued willingness to learn, think, and to explore. No beer until the project is completed and you can turn on a several LEDs…but, it's getting close. Hey, these are all still good.

Musings:

Are cross purposes really angry or just a little upset? If C and C++ support casting integers and floats, why don't they also support casting aspersions? Can fishermen cast aspersions? Why do the same kinds of animals make different sounds in different countries? Do they really learn to speak the local language? If a Chinese rooster comes to the U.S., does it crow in English or Chinese? Which dialect? Does it have an accent? If you told a French dog to lay down, would it understand? What if you told it to sleep? Why is Donald Duck a duck not a drake? Is it possible to defenestrate an elephant? What would happen if an elephant swallowed a mangel wurtzel then sneezed? If water going down a toilet goes anticlockwise south of the equator, do bottles and jars open clockwise?

Is a baby bear running through the forest without the bottom part called a bare minimum?

Is it true that in Portugal they call Portuguese babies, Portugoslings?

Is it easier to get ketchup out of a bottle if you are south of the equator where gravity makes things fall down?

In the autumn, do they call geese feathers Fall down?

If we are in Seattle on a really really tall building and can see all the way to Australia, we see that the birds are all flying upside down.

If a bird is flying south for the winter, when do they decide to turn over and fly upside down? How about flying north in the spring?

Life is a challenge…so many questions…

Cautions and Warnings:

Now that we are working with a CPU, always make certain that the cables connecting the PC to the DE1-SoC board are not twisted and don't have any knots. If they are twisted or tangled, the compiled C instructions might get reversed as they are downloaded into the target and your program may run backwards, the source and destination registers may be reversed, and writes will become reads and reads writes. Also, try to keep your DE1-SoC board lower than your PC thus making it easier for the electrons to get to your board and making the data transfer much faster. However, this will not affect the speed at which your program runs because by that point, the program is already at its destination. Once implemented on the target platform, you can increase its speed performance by selectively applying an FPGA grease to lubricate the appropriate module inputs and outputs. Avoid using too much, however otherwise, you may have to add several NOPs to the end of the program to give it enough time to stop before it runs off the end.

Background:

In the previous projects, we completed the design and integration of memory, a register file, and ALU subsystems and the design of a single cycle CPU.

For this phase of the project, you should ….

- ✓ Have the subsystems and single cycle CPU from your previous projects fully integrated and working as a complete system.
- ✓ Have a working understanding of memory, registers, and busses, how to move data amongst them, and how to control such transfers.
- ✓ Have an understanding of the architecture and basic operation of a pipelined datapath and associated control for both sequential and nonsequential flow through a program. In particular, an understanding of how data forwarding and branches/jumps will interact.
- ✓ Understand how to compile a C code fragment into an assembly level program utilizing the instructions supported by your design; in this case, this will be the MIPS instruction set.
- ✓ Understand how to decode an ISA level instruction into the necessary steps and actions to effect the execution of that instruction in a pipelined context.
- ✓ Understand finite state machines, how to use them, and when not to use them as tools for controlling the behavior of a digital system.
- ✓ Have a working understanding of basic methods of system I/O such as switches and LEDs.

As we noted, most successful designers begin a design with a high-level architecture for the system they are intending to develop. We now refine and elaborate the architecture that we

have discussed in class and introduced in the previous phase of the project in figure 1.0 below.

As noted earlier, this diagram is incomplete and may not necessarily reflect the full capabilities of the available hardware components. Before you begin your design, it is highly recommended that you carefully read sections 4.1-4.4 in the Patterson and Hennessy 4th ed. text posted on the class webpage under documentation and go over the data sheets for your components.

Suggestions of things to think about as you're reading and designing….

- ✓ What happens if an instruction writes back to register 0, which is required to always be 0 regardless of what is written to that register?
- ✓ The ALU and the Memory can each provide values to be written to the register file, and thus, may both be sources of forward information. Can you think of clever ways to be able to combine some of these?
- ✓ The ALU, branch logic, and memory can all require values from the register files, and thus may need to have values forwarded to them.
- ✓ RS, RT, and RD have different meanings in the different instruction formats and types; these must be carefully considered.

Your text may have some answers…

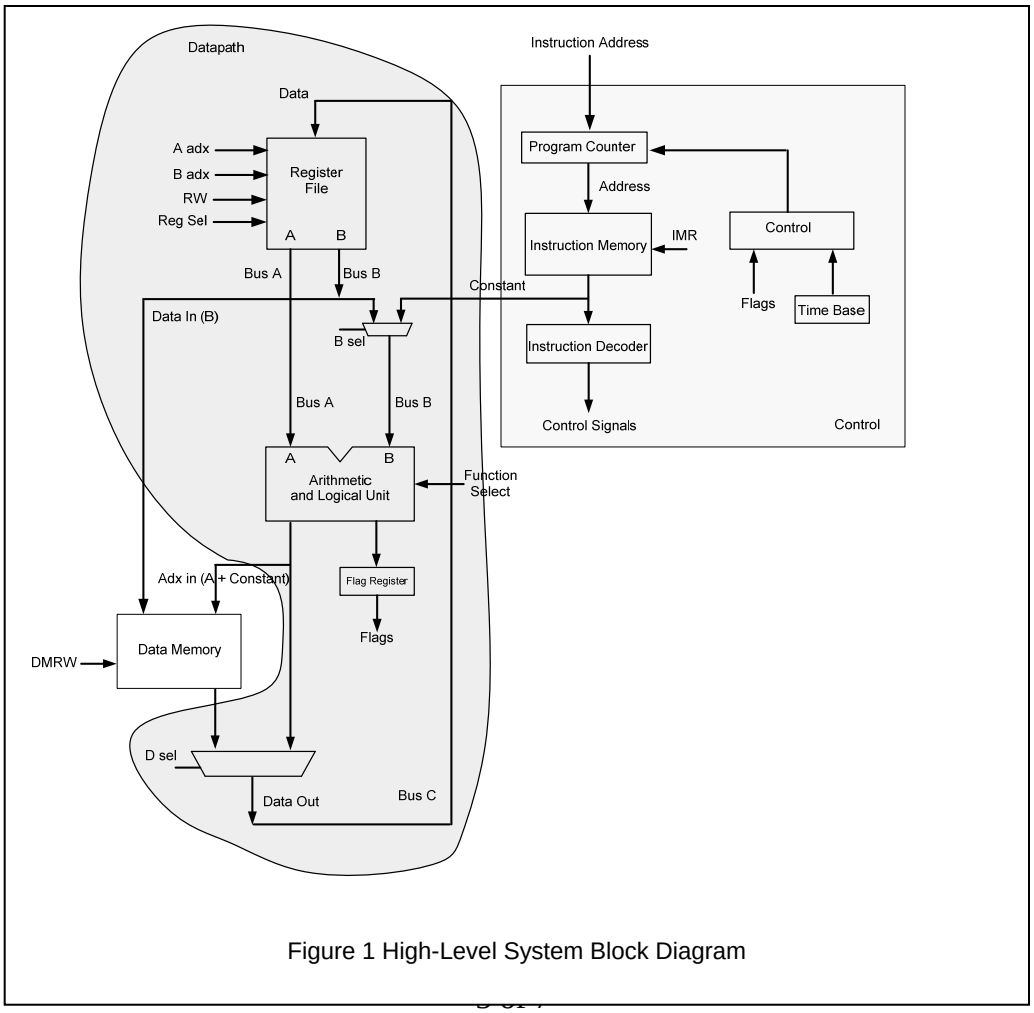

Figure 1 High-Level System Block Diagram

Objectives:

The major objectives of this project include:

- Identifying the control steps and actions necessary to support the following instructions: NOP, ADD, SUB, AND, OR, XOR, SLT, SLL, LW, SW, J, JR, BGT.
- Improving the performance of the single cycle design by incorporating an instruction cycle pipeline.
- Developing the machine code to support control of the pipeline and flow through the datapath to ensure proper execution of each instruction and to manage the data and control hazards.
- Hand compiling a complete simple C code fragment into an assembly level program comprising instructions from the supported instruction set. This means the full code fragment, not its individual C instructions, one at a time.
- Hand compiling the assembly level program into machine code. Again, this means compiling the full assembly level program into the target machine code, not just one instruction at a time.
- Loading the machine code for the complete code fragment, expressed in machine code for your machine, into the instruction memory for your design on the DE1-SoC then executing the complete code fragment, from start to finish.
- Demonstrating the operation of the pipeline and complete instruction cycle for each instruction comprising the code fragment.
- Comparing the performance of the single cycle implementation with that of the pipelined version.

Designing and Implementing the System

1. Based upon the system block diagram given in figure 1.0 above and using the modules and single cycle CPU that you have designed in the previous projects, extend that architecture to support a CPU with a single instruction pipeline.
2. Instruction memory is to be designed and implemented as a 128x32 RAM on the Cyclone V FPGA.
3. Data memory will reside in the SRAM. Words can be stored and accessed as 16 bit quantities for this design, however, they must be sign extended to 32 bits for any operations within the machine.

With the architecture complete, we move to the more detailed design.

Designing the Datapath

1. Modify the single cycle datapath to support the pipeline registers.
2. Modify system busses as appropriate to support point-to-point connections between datapath elements.

Designing the Control

1. Identify the data and control/instruction hazards in your design.
2. Decide how the hazards will be managed.
3. For each of the instructions in the instruction set given in Table 1.0 below, identify the necessary actions and signals to control the elements of the data path to affect the execution of the *fetch, decode, execute, write back,* and *next* phases of the instruction cycle in the pipelined CPU architecture.
4. Identify system constraints that must be considered when designing your control words.
5. Subject to the system constraints, develop your categories of operations.
6. Based upon your categories of operations and system constraints, develop the necessary microinstructions to execute each of the ISA instructions.
7. Identify the control signals necessary to manage the pipeline stages.
8. Design and develop the instruction decoder and the control block to affect the proper flow through the single pipelined instruction cycle.

Compiling and Executing a C Program:

1. For this part of this project, hand compile the complete accompanying C code fragment into MIPS assembly language.
2. Next, using the MIPS instruction set and op codes, hand convert the complete assembly language code fragment into machine code for your CPU.

   It is highly recommended that you carefully go over the MIPS material in the Patterson and Hennessy text and on the class web page as you compile and assemble the code fragment.
3. Load the complete machine code into the instruction memory on the Cyclone V FPGA and the full set of data into the SRAM. These will be our instruction and data memories.

   You are encouraged to look at the test code given in the following directory on the class web page…

   http://www.ee.washington.edu/class/469/peckol/code/HauckTestCode/
4. Load the starting address of the program into the program counter on your machine. Following the basic instruction cycle, *fetch* each instruction from instruction memory, *decode* the op code, *execute* the instruction, *store* any results, determine the address of the *next* instruction, place that address into the program counter and repeat the process until the program terminates.

   You must show that you can execute the full program, from start to finish, without recompiling after executing each instruction.
5. Assign A an initial value of 8 and B an initial value of 3 and repeat steps 1 to 3.

```
int A = 7;
int B = 5;
int C = 3;
int D = 5;
int* dPtr = &D;
unsigned int E = 0x5A5A;
unsigned int F = 0x6767;
unsigned int G = 0x3C;
unsigned int H = 0xFF;

if ((A – B)  > 3
{
      C = C + 4;
      D = C – 3;
      G = E | F;
}
else
{
      C = C << 3;
      *dPtr  = 7;
      G = E & F;
}
A = A + B;
G = (E ^ F) & H;
```

Mnemonic	*Function*
nop	No Operation
add	Add
sub	Subtract
and	Logical AND
or	Logical OR
xor	Exclusive OR
slt	Set Less Than
sll	Shift Left Logical
lw	Load Word
sw	Store Word
j	Jump
jr	Jump Register
bgt	Branch Greater Than

Table 1.0 Instruction Set

Deliverables:

A lab demo showing…

1. A working implementation of a single cycle pipelined CPU design that meets the specified requirements.
2. The complete specified C program compiled into machine code for your machine.
3. The complete set of instructions for the specified program, entered into the instruction memory for your machine.
4. The full set of data for the specified program entered into the data memory for your machine.
5. Once the complete set of instructions and data are entered into the appropriate memories on your CPU, the ability start executing your program on your CPU and have it run to completion.
6. At a minimum, you should be able to show, using Signal Tap, the full instruction cycle for the execution of the specified program, including the values of all variables both during and after execution.

A lab report containing

1. The report must cover the design of the single cycle CPU from Project 4 and the pipelined CPU from Project 5.
2. The annotated Verilog and C source code for all applications / designs both on the DE1-SoC board and on the PC.
3. The annotated assembler and machine code for the code fragment.

4. Representative screen shots showing the results of testing the various designs – from iVerilog or your favourite tool.
5. Representative screen shots showing the Signal Tap logic analyzer outputs for the various designs.
6. Discussion and comparison of the performance of the single cycle architecture from Project 4 with the pipelined version from Project 5.
7. Answers to any questions above.
8. Other things that you deem to be important.
9. Anything that we haven't thought of.