Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2012

Systems@ETH Zürich

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 6: Pipelined Processors

Assigned on: **8th Nov 2012**
Due by: **15th Nov 2012**

1 Paper Exercises

1.1 Data Hazards in Pipelines

There is a data hazard in the following Y86 assembly code. Assume the processor is based on the 5 stage Y86 PIPE- pipeline (without data forwarding).

```
irmovl    $3,   %edx
irmovl    $4,   %eax
subl      %edx, %eax
irmovl    $10,  %edx
addl      %edx, %eax
```

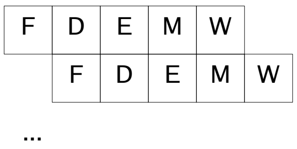
1.1.1 Find the Hazards

Identify and describe all data hazards in the following schema:

F	D	E	M	W					`irmovl $3, %edx`
	F	D	E	M	W				`irmovl $4, %eax`
		F	D	E	M	W			`subl %edx, %eax`
			F	D	E	M	W		`irmovl $10, %edx`
				F	D	E	M	W	`addl %edx, %eax`

1.1.2 Fix the Hazards

Fix the data hazards by inserting the minimal number of NOPS. Use the template below to fill in your solution.



```
irmovl $3, %edx
```

1.2 Control Flow Hazards in Pipelines

Consider the following Y86 assembly code:

```
      irmovl $4,   %esi
      irmovl $1,   %eax
L26:
      irmovl $-2,  %edx
      addl   %esi, %eax
      addl   %edx, %esi
      testl  %esi, %esi
      jne    L26
      irmovl $10,  %ebx
      addl   %ebx, %eax
      halt
```

1.2.1 Perfect Branch Predictor

Resolve the pipeline hazards that might occur through stalling and assume that we have a perfect branch predictor (the processor always knows in advance if the branch is taken or not). How many cycles does the execution of the program take before the 'halt' instruction is fetched? (Once again, the processor has no data forwarding)

1.2.2 Conservative Branch Predictor

Now take the following (more realistic) branch predictor: The predictor is initialized with the assumption that the branch is **not** taken. Whenever a branch occurs, the predictor predicts the same control flow that was executed in the last iteration. Note that for the case that the prediction is detected to be wrong, the pipeline has to be flushed (1 cycle penalty) and refilled (additional penalties). How many cycles does the program take before the 'halt' instruction is fetched?

1.3 Special Control Conditions

Consider the following Y86 assembly code, which is annoted with runtime addresses:

```
     Incr:
0x11      pushl %ebp
0x13      rrmovl %esp, %ebp
0x15      mrmovl 8(%ebp), %eax
0x1b      irmovl $1, %ecx
0x21      addl %ecx, %eax
0x23      rrmovl %ebp,%esp
0x25      popl %ebp
0x27      ret
0x28      irmovl $7, %eax
0x2e      irmovl $8, %ebx
0x34      halt
     Main:
0x35      irmovl  $3, %esi
0x3b      pushl %esi
0x3d      call Incr
0x42      halt
```

- Assume the code is executed by a PIPE processor (i.e., with data forwarding). Describe the pipelined execution of the `Incr` function that is called by `Main` if the processor does not consider special control conditions. Give a schema of the execution similar to Question 1.1.
- Arriving at the `ret` instruction at address `0x27`, what should be the next instruction to be executed? What is the prediction of the PIPE processor?
- Considering special control conditions (such as `ret` instructions), insert the required number of NOPs to avoid fetching wrong instructions. How many cycles does the execution of the program take, starting at address `0x35`, to fetch the final `halt` instruction at address `0x42`?

1.4 Optimization and Performance Measurement

Consider the following functions:

```
int min(int x, int y) { return x < y ? x : y; }
int max(int x, int y) { return x < y ? y : x; }
void incr(int *xp, int v) { *xp += v; }
int square(int x) { return x * x; }
```

The following three code fragments call these functions:

A.

```
for (i = min(x, y); i < max(x, y); incr(&i, 1))
    t += square(i);
```

B.

```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

C.

```
int low = min(x, y);
int high = max(x, y);

for (i = low; i < high; incr(&i, 1))
    t += square(i);
```

Assume x equals 10 and y equals 100. Fill in the following table indicating the number of times each of the four functions is called in the code fragments A-C.

Code	min	max	incr	square
A.				
B.				
C.				

2 Y86 Simulator

2.1 Optimize nadd on PIPE

Information on installing the Y86 tools needed for this assignment can be found in the **CS:APP Guide to Y86 Processor Simulators** available on the course page (`simguide.pdf`) and during the recitation.

2.1.1 Instructions

Download the tarball named `assignment6-sim.tar.gz` from the course web page and extract it using a command such as `tar xvfz assignment6-sim.tar.gz`. A directory called `sim` will appear.

Optimize the given Y86 program with the standard pipelining processor simulator (*PIPE*). The required behavior of this program is defined by the C function `nadd` in `nadd.c` inside `sim/homework/` folder.

The `nadd` function adds a *`len`*-element integer array *`src`* to a non-overlapping array *`dst`*, returning a count of the number of positive integers contained in the result *`dst`*. `nadd.ys` inside `sim/homework/` folder shows the baseline Y86 version of `nadd`. **Your task** is to modify `nadd.ys` with the goal of making `nadd.ys` run as fast as possible on the PIPE simulator.

You can make modifications to `nadd.ys` with the following constraints:

- Your `nadd.ys` function must run correctly. By correctly, we mean that it must correctly add the elements from `src` block to `dst` block *and* return (in `%eax`) the correct number of positive integers in the result `dst` block.
- Your `nadd.ys` function must work for arbitrary array sizes.

Other than that, you are free to make any semantics preserving transformations to the `nadd.ys` function, such as swapping instructions, replacing instructions with other instructions, deleting some instructions, and adding other instructions.

Please read `README` file about how to build the simulator. After building up the simulator, you will work in `sim/homework/` directory for this assignment. In order to test your solution, you will need to build a driver program that calls your `nadd` function. We have provided you with the `gen-driver.pl` program that generates a driver program for arbitrary sized input arrays. For example, typing

```
unix> make
```

inside `sim/homework/` will construct the following two useful driver programs:

- `sdriver.yo`: A *small driver program* that tests a `nadd` function on small arrays with 4 elements. If your solution is correct, then this program will halt with a value of 3 in register `%eax` after adding the elements from `src` array to the `dst` array.
- `ldriver.yo`: A *large driver program* that tests a `nadd` function on larger arrays with 63 elements. If your solution is correct, then this program will halt with a value of 62 (`0x3e`) in register `%eax` after adding the elements from `src` array to `dst` array.

To test your solution in GUI mode on a small 4-element array, type:

```
unix> ../bin/psim -g sdriver.yo
```

To test your solution on a larger 63-element array, type:

```
unix>../bin/psim -g ldriver.yo
```

You can also test your code on a range of block lengths with the ISA simulator. The Perl script `correctness.pl` (in `sim/homework/`) generates driver files with block lengths from 1 up to some limit (default 64), simulates them with `YIS`, and checks the results. It generates a report showing the status for each block length:

```
unix> ./correctness.pl -f nadd.ys
```

2.1.2 Performance Evaluation

The performance of your function is defined in units of *cycles per element* (CPE). That is, if the simulated code requires C cycles to copy a block of N elements, then the CPE is C/N. The PIPE simulator displays the total number of cycles required to complete the program. The baseline version of the `nadd` function running on the PIPE simulator with a large 63-element array requires 1165 cycles to copy 63 elements, for a CPE of $1164/63 = 18.49$.

Since some cycles are used to set up the call to `nadd` and to set up the loop within `nadd`, you will find that you will get different values of the CPE for different block lengths (generally the CPE will drop as N increases). We will therefore evaluate the performance of your function by computing the average of the CPEs for blocks ranging from 1 to 64 elements. You can use the Perl script `benchmark.pl` (in `sim/homework/`) to run simulations of your `nadd.ys` code over a range of block lengths and compute the average CPE. Simply run the command

```
unix> ./benchmark.pl -f nadd.ys
```

to see what happens. For example, the baseline version of the `nadd` function has CPE values ranging between 49.0 and 18.48, with an average of 20.30. Note that `benchmark.pl` does not check for the correctness of the answer. You are expected to achieve an average CPE of at least less than 18.5.

2.2 PIPE-X

Figure 1(a) shows the hardware structure of PIPE, the implementation of the standard pipelined Y86 processor (this is Figure 4.52 in the textbook). Now consider an implementation that cannot include all the bypass paths in PIPE. For the purpose of this assignment, assume that the path *e_valE* is not implemented. Figure 1(b) shows the hardware structure of this machine, called PIPE-X.

a) Determine the detection conditions for the pipeline control logic for PIPE-X. (You can start with the detection conditions for PIPE as shown in Figure 4.64 of the textbook.)

Condition	Trigger

b) Provide a code sequence of Y86 instructions that will incur a stall on PIPE-X but will not include a stall on the PIPE implementation.

c) Rebuild the simulator with *`VERSION=x`* (in `sim/`), you will get the PIPE-X simulator.
```
make VERSION=x
```
Take the `nadd` program you created in previous section. What's the average CPE of your program with the PIPE-X simulator? If the average CPE of your program with PIPE-X and with PIPE differs, try to improve your program further with PIPE-X. Your program should achieve an average CPE of at least less than 18.5 with PIPE-X.

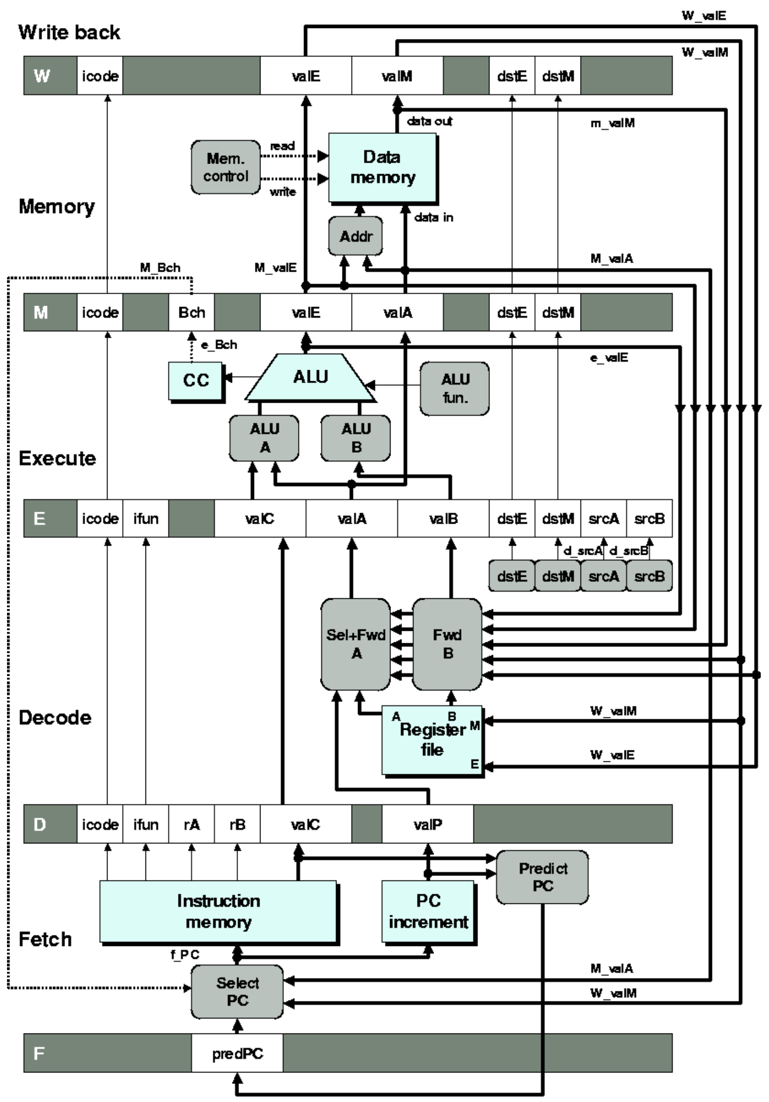
2.3 Hand-In

Hand in the paper exercises to your assistant during the recitation sessions and hand in the simulator exercises via SVN, in the same way as the previous programming assignments.

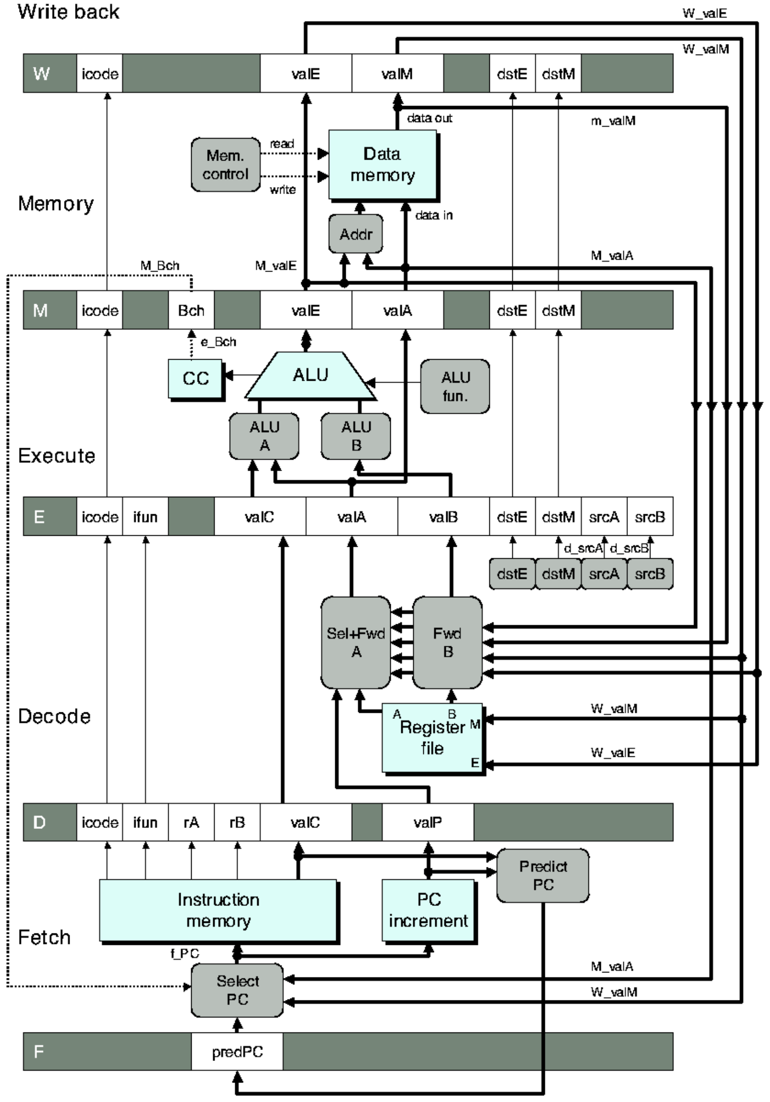
2.4 Hints

The student lab machines (stud[1..26]-h[56..57].inf.ethz.ch) have the necessary packages installed to compile the simulator. If you choose to use your own Linux machine, you will need to install at least the following packages (ask your TA if you have more questions):

- flex
- tcl
- tcl-dev
- tk
- tk-dev
- bison



(a) Hardware structure of PIPE.



(b) Hardware structure of PIPE-X.

Figure 1: Hardware structure of PIPE and PIPE-X.