CSC 631: High-Performance Computer Architecture

Implementing the *Pinkie* Processor in C

Instructor: Haidar M. Harmanani
Due: December 9, 2022

October 14, 2022

1 The Pinkie Machine

The `Pinkie` processor is a 32-bit RISC-based superscalar architecture that is based on the ARM processor. It is a second generation `Pinky` machine that has two cores, and a 256K byte-aligned memory. In order to simplify the simulation, we will be assuming two memories: a *data memory* as well as an *instruction memory*, we will also assume that the memory is 32-bit word aligned. That is, no need to increment the PC by 4 in order to move to the next memory location.

The `Pinkie` processor has two cores and total of 64 general purpose registers, all of which are 16 bits. These include 32 general-purpose integer registers, `W0-W31`, and 32 general-purpose floating-point registers, `F0-F31`. The `Pinkie` processor has also a set of special purpose registers for functions, return values, temporary registers, and global registers, as shown in Table 1. Although not very efficient, the register file is the only set of registers in the pinky machine and will be shared across both cores. Each has core has an integer unit ALUs, a Multiplier, and a floating point ALU.

Figure 1 illustrates the `Pinkie` processor instruction set architecture which are detailed in Table 1. The ISA includes four types of family set ISA:

1. The `D-Type` refer to mostly `load` and `store` operations.
2. The `R-Type` refer to register type instructions.
3. The `I-Type` refers to immediate instructions.
4. The `B-Type` refers to branch instructions

Figure 1: Pinky Machine ISA

The `Pinkie` machine a 4-stage pipeline and includes the following stages: Instruction Fetch (IF), Instruction Decode (ID), Instruction Issue (II), Execute (EX), Memory (M), and Writeback (WB). Assume that all integer instructions have a latency of one, and integer instructions do not contend resources with FP instructions. *Load* and *store* instructions have a latency of three and the `B-Type` which have a latency of two. The floating point adder/subtractor has a latency of at least three, the floating point multiplier a latency of at least six, and the floating point divider has at least a latency of 24. However, your program should allow the user to override these values for simulation purposes. The system can perform concurrently integer operations, floating point additions, floating point multiplications, and floating point divisions.

Programs are stored in the memory and are indexed by the program counter (PC). All instructions are 32 bits in length, and are word aligned. Register `W29` is used as the subroutine link register, and stores the return address when Branch with Link operation is performed. To return from a subroutine, a return instruction would restore the PC from the link register. We will assume that there no exceptions at this stage and we will not handle them. `Pinkie` instructions are made to execute conditionally by postfixing them with the appropriate condition code field:

```
ADDS (W11, W11,W31)  // address 9
BEQ(12)              // address 10
ADD(W12, W13, W14)   // address 11
                     // address 12
```

Data processing instructions do not affect the condition code flags but the flags can be set by using "S". CMP does not need "S". When executing branch instructions and for simplicity, the processor will use the address in the instruction as is. That is, there will be no for any shifting or additions. To ensure that the simulation ends, we will store a halt instruction in the last line of the program. The corresponding datapath for the `Pinkie` processor is shown in Figure 2.

The superscalar `Pinkie` processor supports out-of order execution via a scoreboard. The scoreboard has a number of reservation stations. The number of reservation stations for each functional unit is fully parametrizable, i.e., the number of reservation stations for each FU, and the number of execution cycles for each FU should all be the inputs to your simulator. Instructions will only execute if all of their data dependencies have been resolved, but they may issue in any order.

2 C Elements for an Instruction Set

There are four elements that you would need to implement in order to simulate an instruction set. These are as follows:

1. Opcodes
2. Mnemonics
3. Memory
4. Hardware elements including registers, ALUs, and interconnection components

We will focus on this lab on the design and implementation of a `Pinkie` processor architecture that implements the instruction set shown in Figure 1.

Register	Purpose	Effect or Usage
`F0 - F31`	Floating point registers	
`W0 - W7`	procedure arguments/results	Passing arguments to methods
`W8 - W18`	Temporary registers.	Values will not be saved.
`W19 - W29`	Saved	
`W30`	Link register	`W30` ← PC
`W31`	The constant value 0	

Table 1: Pinkie Processor Registers

OpCode	Instruction	Mnemonic	Effect
0	HALT	HALT	Halts execution
1	LDW	LDW(W_a, address, displacement)	PC++; $W_a \leftarrow$ M[address + displacement]
2	STW	STW(W_a, address, displacement)	PC++; M[address + displacement] $\leftarrow W_a$
3	LDWD	LDWD(W_a, address, displacement)	PC++; $F_a \leftarrow$ M[address + displacement]
4	STWD	STWD(W_a, address, displacement)	PC++; M[address + displacement] $\leftarrow F_a$
5	ADD	ADD(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] + R[W_c]
6	SUB	SUB(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] − R[W_c]
7	ADDD	ADDD(F_a, F_b, F_c)	PC++; R[F_a] = R[F_b] + R[F_c]
8	SUBD	SUBD(F_a, F_b, F_c)	PC++; R[F_a] = R[F_b] − R[F_c]
9	MUL	MUL(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] × R[W_c]
10	MULD	MULD(F_a, F_b, F_c)	PC++; R[F_a] = R[F_b] × R[F_c]
11	DIV	DIV(F_a, F_b, F_c)	PC++; R[W_a] = R[W_b] / R[W_c]
12	DIVD	DIVD(F_a, F_b, F_c)	PC++; R[F_a] = R[F_b] / R[F_c]
13	AND	AND(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] & R[W_c]
14	OR	OR(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] | R[W_c]
15	XOR	XOR(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $\oplus$ R[W_c]
16	ADDS	ADDS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] + R[W_c]; $Z \leftarrow (R[W_a] == 0)$? 1 : 0 $N \leftarrow (R[W_a] < 0)$? 1 : 0
17	SUBS	SUBS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] − R[W_c] $Z \leftarrow (R[W_a] == 0)$? 1 : 0 $N \leftarrow (R[W_a] < 0)$? 1 : 0
18	SL	SL(W_a, W_b, $ShiftAmt$)	PC++; R[W_a] = R[W_b] << ShiftAmt
19	SR	SR(W_a, W_b, $ShiftAmt$)	PC++; R[W_a] = R[W_b] >> ShiftAmt
20	SR	SR(W_a, W_b, $ShiftAmt$)	PC++; R[W_a] = R[W_b] >> ShiftAmt
12–4096	Not Used		

Table 2: Pinkie Processor Instruction Set Summary

3 Deliverables

1. Develop a complete C or Python model for the `Pinky` processor. You will be supplied with a working C skeleton. Complete all missing instructions in the program. That includes the instructions as well as the code. Please also modify the fetch and decode unit. This will include the following:

   (a) Define the Opcodes

```
#define OP_HALT  0x00
#define OP_LDW   0x01
#define OP_STW 0x02
...
#define OP_SR 0xB

#define W0 0x0
#define W1 0x1
#define W2 0x2
...
#define W31 0x1F
```

   (b) Create the Mnemonics

```
#define HALT                                    OP_HALT << 20
#define LDW(DestReg, BaseReg, Address)      (OP_LDW  << 20) | (Address << 10)  | (BaseReg <<
```

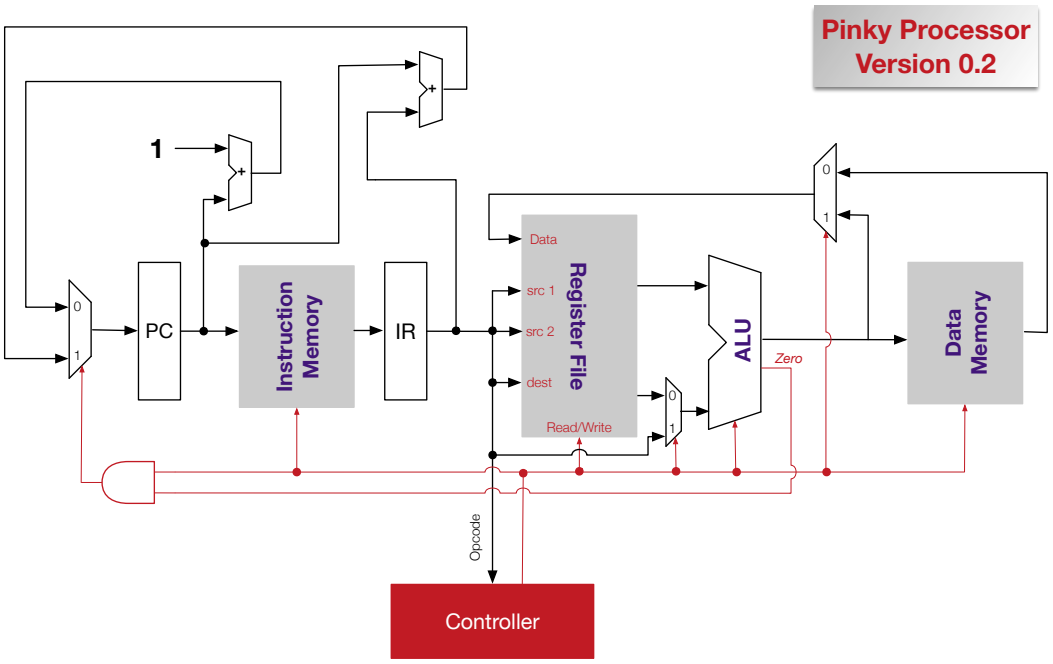

Figure 2: Data path for the Pinky Machine

```
#define STW(SourceReg, BaseReg, Address)   (OP_STW  << 20) | (Address << 10)  | (BaseReg << 5)
#define ADD(DestReg, SourceReg2, SourceReg1) (OP_ADD << 20) | (SourceReg2 << 15) | (SourceReg1
...
...
```

2. All instructions are stored in memory using 16 contiguous bits. Thus, instructions have to be decoded in order to decipher the operations. For example, in order to extract the register index as well as the immediate value from an `I-Type` instruction, one needs to extract the information as follows: `M[iIndex & 0x00FF and (iIndex >> 2) & 0x000F` , respectively.

3. Implement the scoreboard using reservation stations.

4. The input to the simulator should be a text file containing assembly instructions specified in the above format. Read the input file and proceed your simulation with reading the file line by line as if you are fetching binary instructions from an instruction memory. The ID stage is also simplified to parsing the assembly instruction as opposed to decoding the binary. Other inputs are: FU execution cycles as well as their allocated reservation station numbers.

5. Develop a set of programs that test all the implemented operations.

6. Print debugging information to the screen that can help the user debug the memory and the architecture. This includes the reservation station status, instruction status, and the register result status on every cycle.

7. Test your code with a variable number of latencies, instructions, and operations. Report each simulated program along with achieved speed-up.