EECE7352 Computer Architecture
Homework 3

Part A

The ARM Cortex-M85 and the SiFive Essential 7-Series (E7) processors are compared in this section. The Cortex-M85 is a high-performance microcontroller in the ARM Cortex-M family, while the SiFive Essential 7-Series is based on the RISC-V architecture, known for its configurability and scalability in embedded applications.

1. Pipeline Depth
   - The Cortex-M85 features an 8-stage pipeline, optimized for higher clock speeds and the efficient handling of DSP (Digital Signal Processing) and machine learning tasks.
   - The SiFive E7, in contrast, uses a 5-stage pipeline, which prioritizes reduced latency while balancing performance with energy efficiency.

2. Data Path Width
   - The Cortex-M85 has a 32-bit data path, which is designed for effective performance in general embedded processing, DSP, and machine learning workloads.
   - The SiFive E7 series can be configured with either a 32-bit or 64-bit data path, depending on the application's needs. The 32-bit variant is more commonly seen in embedded systems, while the 64-bit option is utilized for higher-performance scenarios.

3. Instructions and Execution Units
   - The Cortex-M85 can execute up to two instructions per cycle, making it a dual-issue processor. It also supports SIMD (Single Instruction Multiple Data) operations, with dedicated execution units that enhance performance for DSP and ML tasks.
   - The E7 is generally a single-issue processor, capable of executing one instruction per cycle. However, additional execution units or features, such as hardware multiply/divide, can be incorporated depending on the configuration. Some versions support RV32IMAC or RV64IMAC extensions, which enable integer multiplication, atomic instructions, and compressed instructions in RISC-V.

4. Branch Prediction and Resolution
   - The Cortex-M85 implements branch speculation and has a dedicated branch resolution unit, enhancing efficiency by reducing delays caused by branches in the pipeline.
   - The E7 typically employs static branch prediction, though more advanced dynamic prediction can be added depending on the specific configuration.

Summary Table:

Characteristic	ARM Cortex-M85	SiFive Essential 7-Series (E7)
Pipeline Depth	8 stages	5 stages
Data Path Depth	32-bit	32-bit or 64-bit
Instruction Issuing	Dual issue (2 per cycle)	Single issue (1 per cycle)
Execution Units	Specialized DSP/ML support	Configurable
Branch Prediction	Branch speculation	Static (can configure as dynamic)

References:

[1] "Documentation – Arm Developer," *developer.arm.com*. https://developer.arm.com/documentation/101924/0100/

[2] "SiFive Essential™," *SiFive*, 2022. https://www.sifive.com/cores/essential

Part B

The first conditional branch predictor uses a 32-entry pattern-based predictor, each entry having a 2-bit counter. The predictor predicts whether a branch is taken or not with the help of a history table. The history table stores the previous two results, i.e. taken or not taken for all addresses. If the previous results are not available, the predictor uses the 2-bit counter. The 2-bit counter represents four prediction states - weakly taken, strongly taken, weakly not taken, and strongly not taken. The counters are initially initialized as weakly taken. The index of the address in the 32-entry table is found by taking the modulus of the address with 32 to restrict it within the range. The accuracy of this branch predictor is 87.0932%. The second predictor also uses a 32-entry predictor with a 2-bit counter. The 2-bit counters are initialized as weakly not taken. It does not use the information of previous branch results to predict whether the current branch is taken or not. The accuracy of this branch predictor is 70.3145%. The third predictor includes pattern history register that records the outcome of the last N branches. To obtain the index of the address within the 32-entry table, the pattern history register is XOR'd with the address. Five different values of N – 2, 4, 8, 16, 32 are implemented. Table 1 shows the results.

Predictor	History Length, N	Number of Correct Predictions	Number of Incorrect Predictions	Accuracy
Predictor 1	-	14297473	2118806	87.0932
Predictor 2	-	11543027	4873252	70.3145
Predictor 3	2	11281760	5134519	68.7230
	4	10975626	5440653	66.8581
	8	10990279	5426000	66.9474
	16	10990279	5426000	66.9474
	32	10990279	5426000	66.9474

Table 1: Conditional Branch Predictor Results

It can be observed that predictor 1 achieves the highest accuracy, making it the best performing model. Predictor 2 has a lower accuracy, indicating it is less effective at predicting branch outcomes since it uses just a 2-bit counter and does not consider the previous history. Predictor 3, which incorporates a pattern history register with different history lengths N, shows the influence of the history length on prediction accuracy. For N values of 2, 4, 8, 16, and 32, the accuracy remains relatively stable, with the highest accuracy for Predictor 3 being 68.72% at N = 2, and the lowest being 66.86% at N = 4. However, increasing N beyond 2 provides no significant improvement, as the accuracy stabilizes around 66.95%. Predictor 1 may have performed the best as it considers the history of the specific branch instruction address rather than considering global addresses like in predictor 3. The python codes for the predictors are available in the folder 'PartB'.

Part C

Problem C.1

```
Loop: ld x1,0(x2)     ;load x1 from address 0+x2
      addi x1,x1,1    ;x1=x1+1
      sd x1,0,(x2)    ;store x1 at address 0+x2
      addi x2,x2,4    ;x2=x2+4
      sub x4,x3,x2    ;x4=x3-x2
      bnez x4,Loop    ;branch to Loop if x4!= 0
```

a) Data Dependencies:
1. `ld x1,0(x2)`: Reads from x2, writes to x1
2. `addi x1,x1,1`: Reads from x1, writes to x1
3. `sd x1,0,(x2)`: Reads from x1, x2
4. `addi x2,x2,4`: Reads from x2, writes to x2
5. `sub x4,x3,x2`: Reads from x3,x2, writes to x4
6. `bnez x4,Loop`: Reads from x4

Register	Source Instruction	Destination Instruction	Type
x1	`ld x1,0(x2)`	`addi x1,x1,1`	Read after Write (RAW)
x1	`addi x1,x1,1`	`sd x1,0,(x2)`	Read after Write (RAW)
x1	`ld x1,0(x2)`	`sd x1,0,(x2)`	Read after Write (RAW)
x2	`addi x2,x2,4`	`sub x4,x3,x2`	Read after Write (RAW)
x4	`sub x4,x3,x2`	`bnez x4,Loop`	Read after Write (RAW)

b) Pipelining:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`ld`	F	D	X	M	W													
`addi`		F	D	s	s	X	M	W										
`sd`			F	s	s	D	s	s	X	M	W							
`addi`						F	s	s	D	X	M	W						
`sub`									F	D	s	s	X	M	W			
`bnez`										F	s	s	D	s	s	X	M	W

The first addi must wait till after the ld gets to the WB stage to obtain the value of x1. The sd must wait till the addi instruction computes the value of x1, i.e. WB. Sub must wait till the second addi

computes the value of x2 i.e. till WB stage. Finally, bnez must wait till sub computes the value of x4.
Number of cycles for one loop iteration = **18 cycles**

Problem C.7

Assume that the original machine is a 5-stage pipeline with a 1 ns clock cycle. The second machine is a 12-stage pipeline with a 0.6 ns clock cycle. The 5-stage pipeline experiences a stall due to a data hazard every five instructions, whereas the 12-stage pipeline experiences three stalls every eight instructions. In addition, branches constitute 20% of the instructions, and the misprediction rate for both machines is 5%.

5-stage pipeline:
- 1ns clock cycle
- 1 stall every 5 instructions (data hazard)
- Misprediction rate: 5%
- 20% instructions are branches

12-stage pipeline:
- 0.6ns clock cycle
- 3 stalls every 8 instructions (data hazard)
- Misprediction rate: 5%
- 20% instructions are branches

a) What is the speedup of the 12-stage pipeline over the 5-stage pipeline, taking into account only data hazards?

5-stage pipeline:

$$Cycles\ per\ instruction, CPI_{data,5-stage} = 1 + \frac{1}{5} = 1.2$$

$$Execution\ Time, ET_{5-stage} = CPI_{data,5-stage} * Clock\ Cycle\ Time_{\ 5-stage} = 1.2 * 1ns = 1.2ns$$

12-stage pipeline:

$$Cycles\ per\ instruction, CPI_{data,12-stage} = 1 + \frac{3}{8} = 1.375$$

$$Execution\ Time, ET_{12-stage} = CPI_{data,12-stage} * Clock\ Cycle\ Time_{\ 12-stage} = 1.375 * 0.6ns = 0.825ns$$

Speedup of the 12-stage pipeline over the 5-stage pipeline is given by:

$$Speedup = \frac{Execution\ Time, ET_{5-stage}}{Execution\ Time, ET_{12-stage}} = \frac{1.2}{0.825} = \mathbf{1.454}$$

b) If the branch mispredict penalty for the first machine is 2 cycles but the second machine is 5 cycles, what are the CPIs of each, taking into account the stalls due to branch mispredictions?

Probability of misprediction rate is given as:

$$Branch\ Misprediction\ Rate = 0.05 * 0.2 = 0.01$$

Calculating cycles per instruction due to branch mispredictions:

$$CPI_{branch,5-stage} = 0.01 * 2 = 0.02$$

$$CPI_{branch,12-stage} = 0.01 * 5 = 0.05$$

Total CPI:

$$CPI_{5-stage} = CPI_{data,5-stage} + CPI_{branch,5-stage} = 1.2 + 0.02 = \mathbf{1.22}$$

$$CPI_{12-stage} = CPI_{data,12-stage} + CPI_{branch,12-stage} = 1.375 + 0.05 = \mathbf{1.425}$$

Problem 3.1

Given Latency cycles for each instruction,

ld	3
sd	1
add / sub	0
Branches	1
fadd.d	2
fmul.d	4
fdiv.d	10

Loop:	fld	f2,0(Rx)	1 + 3
	fmul.d	f2, f0, f2	1 + 4
	fdiv.d	f8, f2, f0	1 + 10
	fld	f4, 0(Ry)	1 + 3
	fadd.d	f4, f0, f4	1 + 2
	fadd.d	f10, f8, f2	1 + 2
	fsd	f4, 0(Ry)	1 + 1
	addi	Rx, Rx, 8	1

addi	Ry, Ry, 8	1
sub	x20, x4, Rx	1
bnz	x20, Loop	1 + 1
Cycles per loop iteration		**37**

Problem 3.2

```
Loop: fld f2,0(Rx)
<stall due to ld>
<stall due to ld>
<stall due to ld>
I0: fmul.d f2,f0,f2
<stall due to mul>
<stall due to mul>
<stall due to mul>
<stall due to mul>
I1: fdiv.d f8,f2,f0
I2: fld f4,0(Ry)
<stall due to ld>
<stall due to ld>
<stall due to ld>
<stall due to div>
<stall due to div>
<stall due to div>
<stall due to div>
<stall due to div>
I3: fadd.d f4,f0,f4
I4: fadd.d f10,f8,f2
I5: fsd f4,0(Ry)
I6: addi Rx,Rx,8
I7: addi Ry,Ry,8
I8: sub x20,x4,Rx
I9: bnz x20,Loop
<stall for branch delay>
```

Cycles per loop iteration = **27**

Problem 3.3

	Execution pipe 0	Execution pipe 1
Loop:	fld f2,0(Rx)	<nop>
	<stall due to ld>	<nop>
	<stall due to ld>	<nop>
	<stall due to ld>	<nop>
	fmul.d f2, f0, f2	<nop>
	<stall due to mul>	<nop>
	<stall due to mul>	<nop>
	<stall due to mul>	<nop>
	<stall due to mul>	<nop>
	fdiv.d f8, f2, f0	fld f4, 0(Ry)
	<stall due to ld>	<nop>
	<stall due to ld>	<nop>
	<stall due to ld>	<nop>
	<stall due to div>	<nop>
	<stall due to div>	<nop>
	stall due to div>	<nop>
	<stall due to div>	<nop>
	<stall due to div>	<nop>
	<stall due to div>	<nop>
	<stall due to div>	<nop>
	fadd.d f4, f0, f4	fadd.d f10, f8, f2
	fsd f4, 0(Ry)	addi Rx, Rx, 8
	addi Ry, Ry, 8	sub x20, x4, Rx
	bnz x20, Loop	<nop>
	<stall due to bnz>	<nop>

Cycles per loop iteration = **25**

Problem 3.18

Branch-Target Buffer Processor:
Deeply pipelined processor:
Misprediction penalty: 4 cycles
Buffer miss penalty: 3 cycles
Hit rate: 90%
Accuracy: 90%

Branch frequency: 15%

Other Processor:
Fixed branch penalty: 2 cycles
Base CPI without branch stalls: 1

CPI Calculation for Branch-Target Buffer Processor:

$$CPI_{BTB} = 1 + P_{branch}\left[(P_{hit} * P_{correct} * 0) + \left(P_{hit} * (1 - P_{correct}) * P_{misprediction}\right) + \left((1 - P_{hit}) * P_{miss}\right)\right]$$

Here,
$P_{branch} = 0.15$
$P_{hit} = 0.9$
$P_{correct} = 0.9$
$P_{misprediction} = 4$
$P_{miss} = 3$

$$CPI_{BTB} = 1 + 0.15 * [(0.9 * 0.9 * 0) + (0.9 * 0.1 * 4) + (0.1 * 3)]$$
$$= 1 + 0.15 * [0 + 0.36 + 0.3] = 1 + 0.15 * 0.66 = 1 + 0.099 = \mathbf{1.099}$$

CPI Calculation for Processor with Fixed Branch Penalty:

$$CPI_{fixed} = 1 + P_{branch} * Penalty$$

Here,
Penalty = 2 cycles

$$CPI_{fixed} = 1 + 0.15 * 2 = \mathbf{1.3}$$

Performance Ratio Calculation:

$$Ratio = \frac{CPI_{fixed}}{CPI_{BTB}} = \frac{1.3}{1.099} \approx \mathbf{1.183}$$

∴ The processor with the branch target buffer is 1.18 times faster than the processor with a fixed two-cycle branch penalty.

Part D

The following is x86 assembly for the foo function. Assume that the first operand is a source operand, and the second operand is the destination operand (for ADD, the second operand is both a source and a destination). Write equivalent code in C and RISC-V assembly. For C code, include a mapping between the variables you use and the x86 registers; for RISC-V code, include a mapping between the RISC-V registers you use and the x86 registers. Your code should run on the BRISC-V simulator. Remember to only use the RV32I instruction subset.

```
      .text
      .globl foo
foo:
      movl $0x0, %eax
      movl $0x63, %ebx
      movl $0x63, %ecx
tloop: add %ebx, %eax
            decl %ebx
            decl %ecx
here: cmpl $0x0, %ecx
      jg tloop
ret
```

C code:

```
int foo() {
    int eax = 0;
    int ebx = 0x63;
    int ecx = 0x63;
    while (ecx > 0) {
        eax += ebx;
        ebx--;
        ecx--;
    }
    return eax;
}
```

RISC-V Code:

```
    .text
    .globl foo
foo:
    li t0, 0            # eax = 0
    li t1, 0x63         # ebx = 0x63
    li t2, 0x63         # ecx = 0x63

loop:
    beq t2, zero, end   # if ecx == 0, exit the loop
    add t0, t0, t1      # eax += ebx
    addi t1, t1, -1     # ebx--
    addi t2, t2, -1     # ecx--
    j loop

end:
    mv a0, t0
    ret

main:
    call foo
    jr  ra
    .size   main, .-main
    .ident  "GCC: (GNU) 7.2.0"
```