4190.308-002: Computer Architecture
Midterm Exam
October 28th, 2019
Professor Jae W. Lee

Student ID #: _____

Name: _____

This is a closed book, closed notes exam.
80 Minutes
14 Pages
(+ 4 Appendix Pages)

Notes:

- Please turn off all of your electronic devices (phones, tablets, notebooks, netbooks, and so on). A clock is available on the lecture screen.
- Please stay in the classroom until 30 minutes before the end of the examination.

- You must not discuss the exam's contents with other students during the exam.
- You must not use any notes on papers, electronic devices, desks, or part of your body.

- "**RISC-V Reference sheet**" is provided at the end of this exam; use it as you need.

(This page is intentionally left blank. Feel free to use as you like.)

Part A: Short Answers (20 points)

Question 1 (20 points)

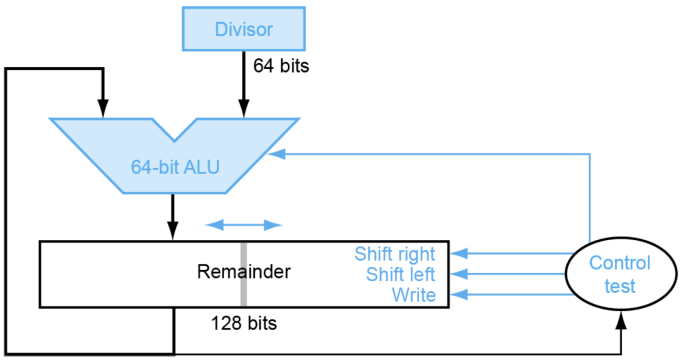
아래 문제에 대해 True/False로 답하시오. 답을 설명할 필요는 없으며, 정답은 각각 4점 부여, 오답은 4점 감점함.

(1) 최신 CPU는 주로 clock frequency의 스케일링으로 성능향상을 달성하고 있다.

(2) IEEE 754 floating-point 연산에서 finite x값과 positive infinity ($+\infty$)를 곱하면 NaN (Not-a-Number)가 된다.

(3) RISC-V ISA에서 temporary registers (t0-t6)은 callee-saved 레지스터이다.

(4) 아래 그림은 수업에서 다룬 optimized integer divider를 보여준다. Division이 끝난 후에, 몫(quotient)은 128비트 Reminder 레지스터의 상위 64비트에, 나머지(reminder)는 하위 64비트에 담겨진다.



(5) 최신 CPU에서는 precise exception을 구현하는데 드는 비용을 줄이기 위해, imprecise exception을 널리 채택하고 있다.

Part B: Instruction Set Architecture (55 points)

Question 2 (25 points)

다음은 (π)의 값을 구하는 식이다.

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} \cdots.$$

아래는 위 식을 구현한 C 코드와 RISC-V 어셈블리 코드이다. 이 함수는 Pi와 M_PI 사이의 오차가 DIFF보다 작아질 때 까지 걸리는 루프의 반복 횟수(iteration count)를 반환한다. RISC-V 어셈블리 코드의 각각의 빈칸을, 주석(comment)과 일치하도록 채워 넣으시오.

```
#define M_PI 3.141592
#define DIFF 0.005
int func()
{
      int i = 1;
      float Pi = 0;
      while(1){
            if(i % 2)    Pi += 4.0 / (2 * i - 1);
            else         Pi -= 4.0 / (2 * i - 1);
            if((M_PI - Pi) < DIFF && (M_PI - Pi) > (0 - DIFF)) break;
            i++;
      }
      return i;
}
```

<PI equation iteration count: C code>

(pseudo instructions LA: load address, LI: load immediate)

```
.data
m_pi: .float      3.141592
diff: .float      0.005
.text
func: LA          t1, diff
      FLD         f0, 0(t1)          # f0: diff
      LA          t1, m_pi
      FLW         f3, 0(t1)          # f3: M_PI
      FMV.D.X     x0, f2       # f2: p = 0
      FADD.Df7, f2, f2         # f7: const 0.0
      FSUB.Df1, f2, f0         # f1: 0 - diff
      LI          t0, 1              # t0: i = 1
      LI          t1, 2              # t1: const 2
      LI          t2, 4
      FMV.D.X     t2, f4       # f4: const 4.0
      FCVT.D.L    f4, f4
L1:   MUL         t2, t0, t1         # t2: (2 * i)
      ADDI        t2, t2, -1         # t2: (2 * i - 1)
      FMV.D.X     t2, f5       # f5: (float) (2 * i - 1)
      FCVT.D.L    f5, f5
      ①_________________           # f6: (4.0 / (2 * i - 1))

      REM         t2, t0, t1         # t2: (i % 2)
      BEQ         t2, x0, L2         # if(i % 2)
      ②_________________           # f2: p += 4.0 / (2 * i - 1)
      J           L3
L2:   ③_________________           # f2: p -= 4.0 / (2 * i - 1)
L3:   ④_________________           # f5: (M_PI - p)
      ⑤_________________           # if((M_PI - p) < diff)
      ⑥_____________ L4
      ⑦_________________           # if((M_PI - p) > (0 - diff))
      ⑧_____________ L5
L4:   ADDI        t0, t0, 1          # t0: i++
      J           L1
L5:   ADD         v0, t0, x0         # return i
      JR          ra
```

<PI equation iteration count: RISC-V assembly code>

Question 3 (30 points)

다음은 특정 C코드에 대한 RISC-V 어셈블리 코드이다. 스택 포인터는 0x70000038을 가리키고 s0레지스터에는 0이 저장되어 있다.

Address	Code
	.text
0x00400024	main: ADDI a0, zero, 5
0x00400028	JAL foo
0x0040002c	J result
0x00400030	foo: ADDI sp, sp, -16
0x00400034	SD ra, 0(sp)
0x00400038	SD s0, 8(sp)
0x0040003c	ADD s0, a2, x0
0x00400040	BEQ s0, x0, return0
0x00400044	ADDI a0, s0, -1
0x00400048	JAL foo # HERE!!
0x0040004c	MULT a0,a0,s0
0x00400050	exitfoo: LD ra,0(sp)
0x00400054	LD s0,8(sp)
0x00400058	ADDI sp,sp,16
0x0040005c	JR ra
0x00400060	return0: LI a0,1
0x00400064	J exitfoo
0x00400068	result: NOP

(1) “HERE!!”이라고 태그 된 jal 명령어가 3번 실행된 직후, stack에 저장된 값을 아래 표에 채워 넣어라.

Stack Address	Value
0x70000038	(Some val.)
0x70000030	
0x70000028	
0x70000020	
0x70000018	
0x70000010	
0x70000008	
...	

(2) 해당 프로그램을 C 코드로 작성하라. 이 프로그램이 끝난 후 리턴 값은 무엇인가? (리턴 값 = a0 레지스터에 저장된 값)

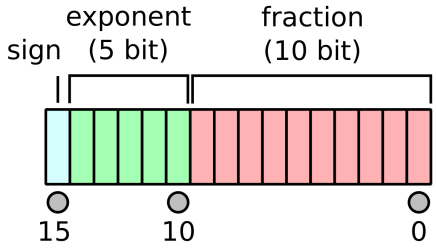
Part C: Floating point (20 points)

Question 4 (20 points)

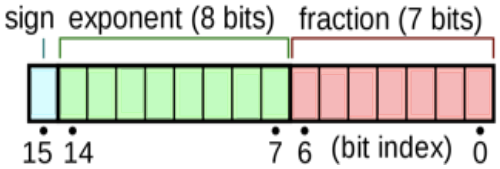
Floating point(FP) 표현은 과학계산, 머신러닝등의 응용에서 널리 사용된다. 모든 워크로드가 32비트 single-precision FP(FP32)를 필요로 하지는 않으므로, 최근 16비트 FP 표현도 널리 사용되고 있다. 이에 대해 알아본다.

(1) FP32 표현된 Num = 0xc0500000 이라할 때, Num의 값을 10진수로 구하시오.

(2) Half-precision FP(FP16)이 아래와 같은 포맷으로 주어질 때(exp bias=15), 위의 Num(=0xc0500000)을 truncate하여 16비트 hexadecimal로 표현하시오.



(3) Brain FP(BFloat16)이 아래와 같은 포맷으로 주어질 때(exp bias=127), 위의 Num(=0xc0500000)을 truncate하여 16비트 hexadecimal 표현하시오.

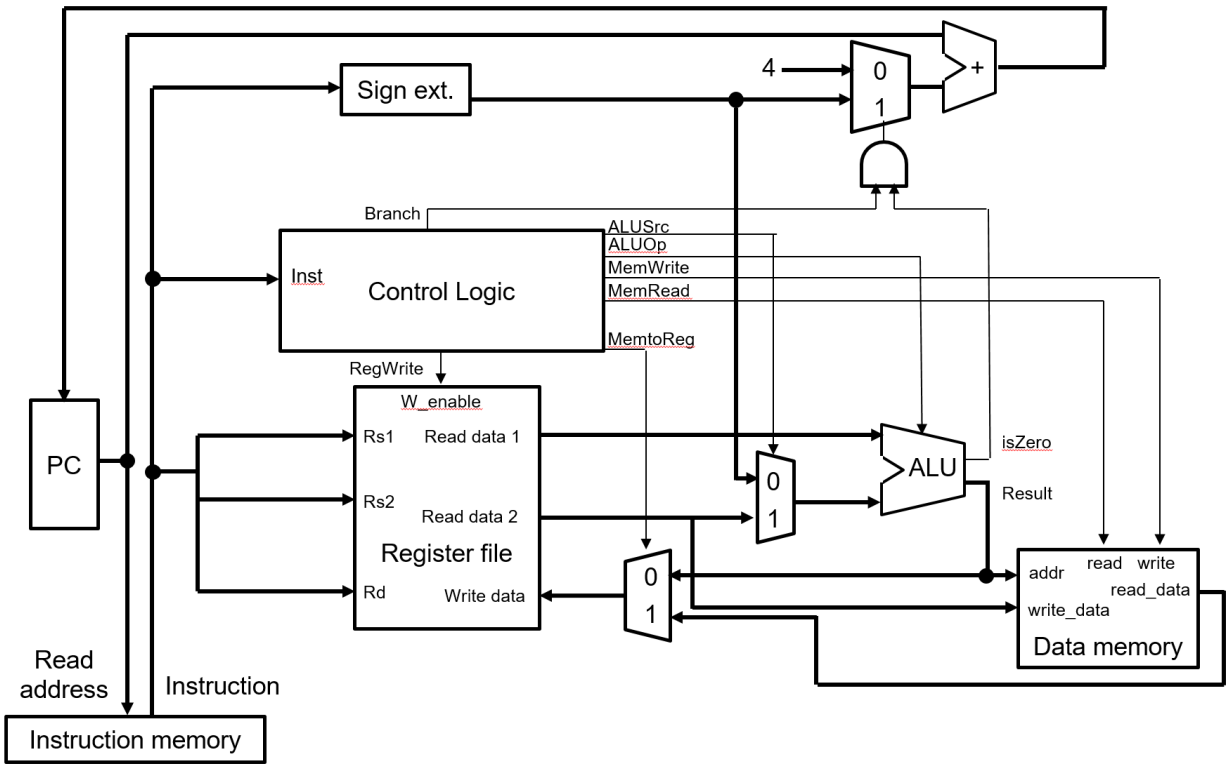


(4) FP16과 bfloat16의 장단점을 서술하라. 그리고, FP32와 mixed-precision으로 연산할 때 유리한 표현은 어느 것인가?

Part D: Single-Cycle RISC-V CPU (45 points)

Question 5 (45 points)

다음 그림은 Single-cycle RISC-V 프로세서를 나타낸 것이다. 다음의 질문에 답하여라.



(1) 다음 표는 각각의 동작을 수행하는 데에 필요한 사이클 수를 나타낸 것이다. 이 프로세서의 최대 동작 가능 클럭 주파수를 구하시오. (단, 표에 없는 동작에 걸리는 시간은 모두 0으로 계산한다)

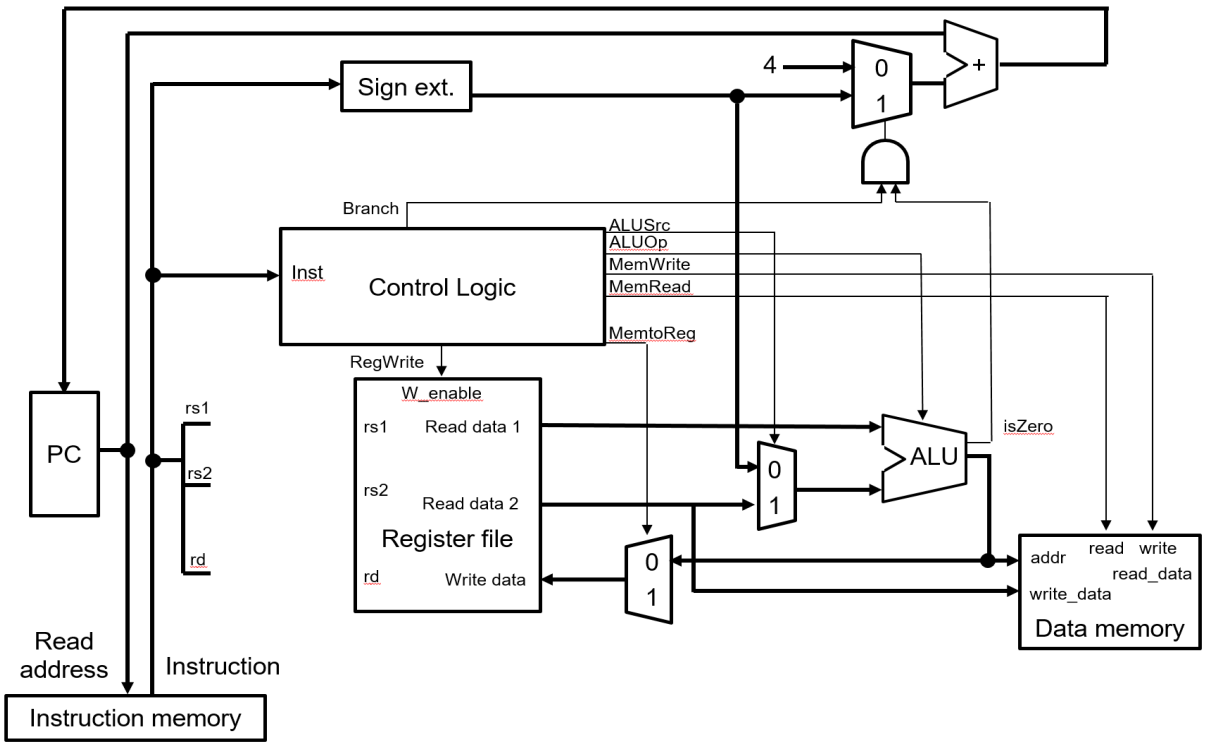
동작	레지스터 읽기/쓰기	명령어 읽기	ALU 연산	메모리 접근
실행 시간(ps)	50 ps	100 ps	100 ps	200 ps

(2) CPU에 새로운 명령어 'addmem'을 추가하려고 한다. 이 명령어는 I-format 명령어이며, 레지스터와 메모리의 값을 각각 읽어서 더하고, 그 값을 다시 레지스터에 쓴다. 단, decoding logic에서 rd의 값을 읽어올 때에는 rs2를 사용한다.

addmem rd, imm(rs1): R[rd] = R[rd] + M[R[rs1] + imm]

해당 명령어를 구현하기 위해 필요한 요소와 제어 신호를 아래 그림에 추가하여라.

Note: 필요하다면 부록 B의 답안지를 사용하시오.



(3) 새롭게 구현한 프로세서에서의 제어 신호에 대한 아래의 표를 완성하여라.

Inst.	ALUOp	ALUSrc	RegWrite	MemWrite	MemRead	MemtoReg	Branch	New signal (from (2))
Add								
Addmem								

*ALUOp에는 다음 중 하나를 사용할 수 있다. (And, Or, Nor, Xor, Add, Sub)

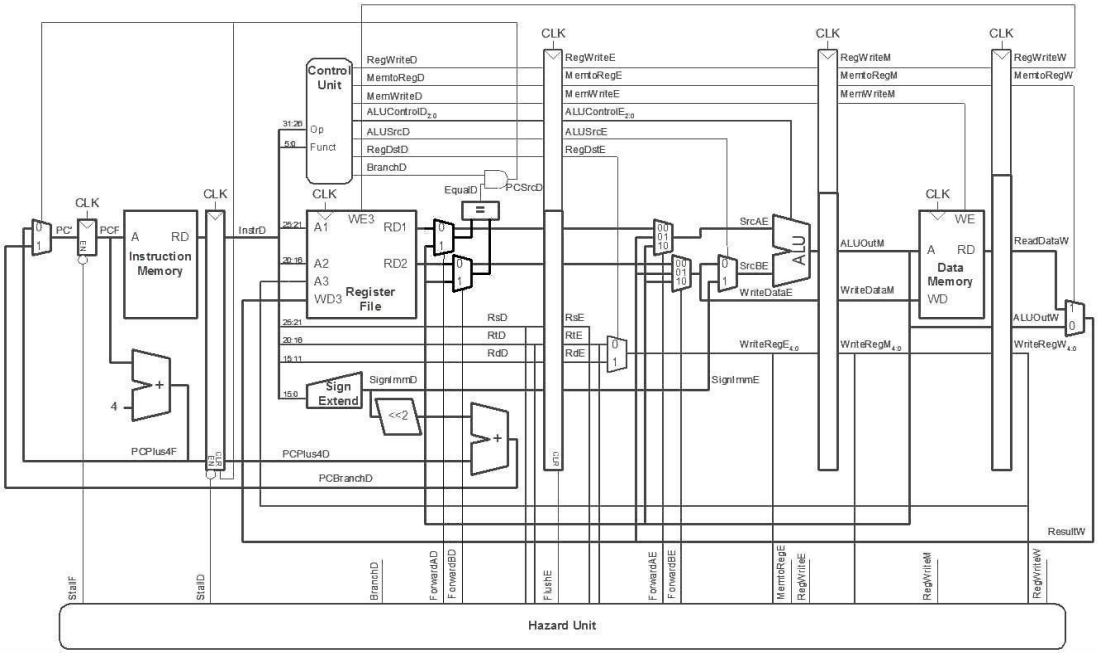
(4) 새롭게 구현한 프로세서의 최대 동작 클럭 주파수를 구하시오. 각각의 구성 요소의 작동에 필요한 시간은 소문제(1)과 같다.

(5) Addmem 명령어를 추가함으로써 총 명령어 개수를 10% 줄일 수 있다고 한다. 이 명령어를 추가함으로써 성능 이득을 볼 수 있는가? 이유와 함께 서술하시오.

Part E: Pipelining (60 points)

Question 6 (30 points)

Bentel 사는 RISC-V 호환 프로세서를 생산하는 회사이다. 이 회사의 제품군으로는 고성능 B7^{TM} 계열 프로세서와 저가형인 B3^{TM} 계열 프로세서가 있다. 다음 그림은 Forwarding logic이 완전히 구현되어 있는 B7 ^{TM} 프로세서의 파이프라인을 나타낸 것이다.



Bentel's high-end b7^{TM} processor pipeline

다음 코드를 사용하여 두 프로세서의 성능을 비교하려고 한다.

```
I1: LD  t1, 40(s1)
I2: ADD         t2, t2, a0
I3: ADD         t3, t1, s0
I4: BEQ         t3, t3, I6    # branch to I6 if taken
I5: LD  t2, 0(t1)
I6: SD  t2, 20(s1)
```

(1) b3^{TM} 프로세서는 다음과 같은 기능을 가지고 있다.

- 표준 5 스테이지 (IF, ID, EX, Mem, WB) 파이프라인
- forwarding logic 없음.
- Data, control hazards에서는 파이프라인을 stall.
- 분기 검사는 두 번째 단계 (ID)에서 일어남.
- 분기 검사가 끝날 때까지 명령어가 fetch되지 않음.

- 같은 클럭 사이클에서 앞의 절반 싸이클에서는 레지스터 쓰기, 뒤의 절반 싸이클에는 레지스터 읽기가 가능

다음 실행 결과표를 채우고, 마지막 명령어 (I6)가 완전히 실행되기까지 필요한 사이클 수를 구하여라. (F: Fetch, D: Decode, E: Execute, M: Memory, W: Write Back).

Notes: 실행되지 않은 명령어는 표시하지 말 것. 필요하다면 부록 C의 답안지에 작성하시오 (별도 표기 바람).

Cycle / Instr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	F	D	E	M	W										

(2) 고성능의 b7™ 프로세서는 (a) full forwarding logic을 포함하고 있으며, (b) 완벽한 분기 예측 기능을 가지고 있다. 다음의 실행 결과표를 완성하고 마지막 명령어 (I6)가 완전히 실행되기까지 필요한 사이클 수를 구하여라.

Cycle / Instr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	F	D	E	M	W										

Question 7 (30 points)

어떤 RISC-V 파이프라인 설계자가 새로운 구조의 도입을 제안하였다. 이 구조는 EX 스테이지에 1비트 플래그 레지스터를 추가하고, SLT (Set-Less-Than) 명령어의 실행 결과에 의해 플래그의 값이 설정된다. 또한, 이 플래그 값을 이용해 조건부 실행 명령어(predicate instruction)를 추가한다. 그가 제안한 구조의 세부 내용은 다음과 같다.

SLT rd, rs1, rs2은 rs1 < rs2이면 플래그 레지스터를 1로, 아니면 플래그를 0으로 설정한다. 그 외의 명령어들은 플래그의 값을 바꾸지 않는다. R-format 명령어 뒤에 suffix ".T"(True)가 붙은 명령어는 플래그가 1일 때만 실행된다. Suffix ".F"(False)가 붙은 R타입 명령어는 플래그의 값이 0일 때만 실행된다.

(1) 다음 어셈블리 코드는 최대공약수를 구하는 코드의 일부이다. 새로운 명령어를 이용해, 최소한의 명령어로 "with predicate" 코드를 작성하여라.

	#without predicate instructions	#with predicate instructions

//lhs != rhs assured
do {
 	
  if(lhs < rhs)
  rhs -= lhs;
  else
  lhs -= rhs;
 	
}while (lhs != rhs);
	…
	L1:
	BLT x2, x3, L2
	SUB x3, x3, x2
	BNE x2, x3, L1
	BEQ x0, x0, exit # taken
	L2:
	SUB x2, x2, x3
	BNE x2, x3, L1
	exit:
	…
		…
		L1:
		SLT x1, x2, x3
		
		BNE x2, x3, L1
		…

(2) 위의 루프가 충분히 많이 반복될 때, 두 코드의 평균 cycles/iteration을 구하여라. If 문이 then clause와 else clause를 실행할 확률은 각각 50-50이며, 모든 branch 명령어는 평균 1.5 cycle의 bubble을 발생시킨다고 가정한다.

(3) Predicate 명령어의 장단점을 간단히 설명하시오(1문단).

Appendix A: RISC-V Reference Sheet (Page 1)

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

RISC-V **Reference Data** ①

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'b M[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'b M[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'b M[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0, +0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fnmsub.s,fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (~F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert to SP from DP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert to DP from SP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

Appendix A: RISC-V Reference Sheet (Page 2)

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

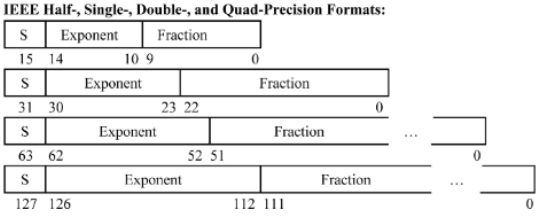
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

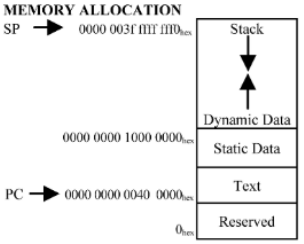
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

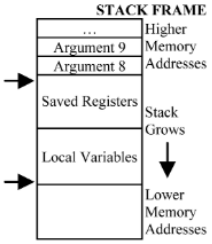
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME

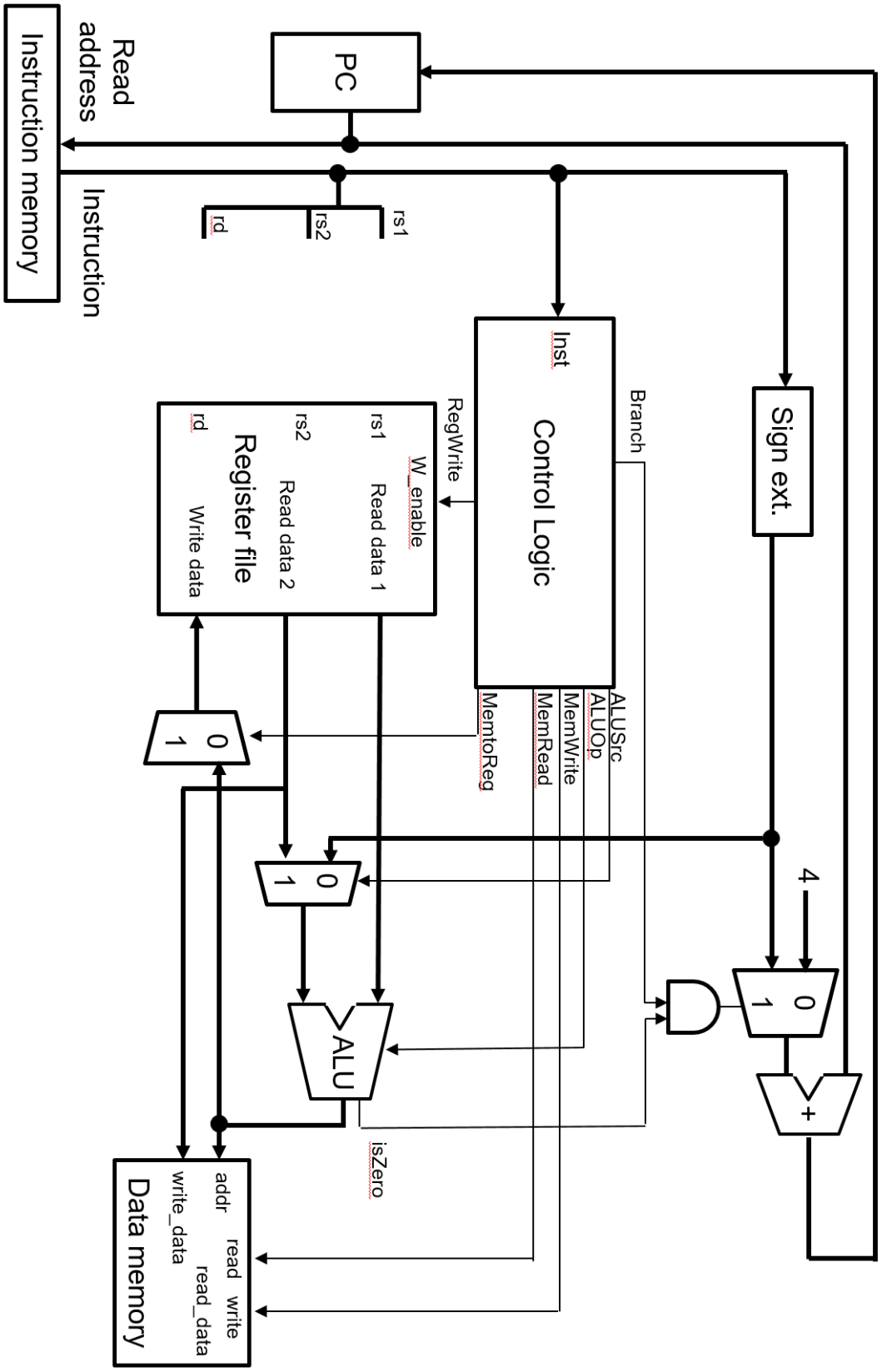


SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

Appendix B: Answer sheet for Question 5



Appendix

Appendix C: Answer sheet for Question 6

Question 6(1) – b3™ Processor

Cycle/ Instr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	F	D	E	M	W										

Question 6(2) – b7™ Processor

Cycle/ Instr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	F	D	E	M	W										