版本

更新记录	文档名	实验指导书_lab3
	版本号	0.3
	创建人	计算机组成原理教学组
	创建日期	2022/1/1

更新历史

序号	更新日期	更新人	版本号	更新内容
1	2022/1/1	陈颖琪	0.1	初版，单周期 CPU 实验
2	2022/4/7	陈颖琪	0.2	修订表3
3	2022/4/19	陈颖琪	0.3	实验报告要求里增加拓展思考问题

文档错误反馈:

Yingqichen@sjtu.edu.cn

1 实验三 简易单周期 CPU 实验

RISC-V 架构 CPU 的传统流程可分为取指、译码、执行、访存、回写 (Instruction Fetch, Decode, Execution, Memory Request, Write Back)，五阶段。

实验一完成了ALU 设计并掌握了存储器IP 的使用；实验二实现了单周期 CPU 的取指、译码阶段，完成了 PC、控制器的设计。在实验一与实验二的基础上，单周期 CPU 的设计的各模块已经具备，再引入数字逻辑课程中所实现的多路选择器、加法器等门级组件，通过对原理图的理解，分析单条（单类型）指令在数据通路中的执行路径，依次连接对应端口，即可完成单周期 CPU。

本次实验要求设计支持22条RISC-V指令的简化版32bit RISC-V CPU核。支持指令：*add，sub，and，or，xor，sll，srl，sra，addi，andi，ori，xori，slli，srli，srai，lw，jalr，sw，beq，bne，lui，jal*。

在进行本次实验前，你需要具备以下基础能力：

1. 熟悉Vivado 的仿真功能(行为仿真)
2. 理解数据通路、控制器的信号

1.1 实验目的

1. 掌握不同类型指令在数据通路中的执行路径。
2. 掌握Vivado 仿真方式。

1.2 实验设备

1. 计算机 1 台 (尽可能达到 8G 及以上内存)；
2. Xilinx Vivado 开发套件(2020.2 版本)。

注：本次实验为 CPU 软核实验，不涉及开发板外围设备，故不需要开发板。

1.3 实验任务

1.3.1 实验要求

阅读实验原理实现以下模块：

(1) Datapath，其中主要包含 alu(实验一已完成)，PC、immext、adder、mux2（adder、mux2 数字逻辑课程已实现)、registerfile（已提供参考代码）。
(2) Controller(实验二已完成)。
(3) 指令存储器 inst_mem(Single Port Ram)，数据存储器 data_mem(Single Port Ram)；使用 Block Memory Generator IP 构造指令，注意考虑 PC 地址位数统一。(参考实验一)
(4) 参照实验原理，将上述模块依指令执行顺序连接。实验给出 top 文件，需兼容 top 文件端口设定。
(5) 实验给出仿真程序，最终以仿真输出结果判断是否成功实现要求指令。

1.3.2 实验步骤

1. 从实验一中，导入 alu 模块；
2. 从实验二中导入PC、sc_cu、immext模块；
3. 从本实验已给源代码中导入多路选择器、加法器模块；
4. 使用Block Memory，例化lpm_rom_irom，使用实验二产生的测试汇编代码段的指令机器码coe 文件；
5. 参考实验原理，连接各模块；
6. 导入顶层文件及仿真文件，运行仿真；

1.4 实验环境

|–sc_computer.v	设计顶层文件，已提供。
|—sc_cpu.v	RISC-V软核顶层，连接CPU核数据通路和控制模块，已提供接口定义，自行完整实现
|——sc_cu.v	控制器模块，使用实验二代码。
|——dff32.v	PC模块，使用实验二代码
|——alu.v	ALU 模块，使用实验一代码
|——immext.v	有符号立即数扩展模块，使用实验二代码
|——mux2x32.v	二选一选择器，已提供
|——mux4x32.v	四选一选择器，已提供
|——regfile.v	寄存器堆，已提供
|——cla32.v	加法器，已提供
|—sc_instmem.v	指令ROM，例化lpm_rom_irom.ip模块，已提供
|—— *lpm_rom_irom.ip*	*单口ROM IP，自行通过Block memory generator 进行实例化*
|—sc_datamem.v	数据RAM，例化lpm_ram_dq_dram.ip模块，已提供
|—— *lpm_ram_dq_dram.ip*	*单口RAM IP，自行通过Block memory generator 进行实例化*
|TB_sc_computer.v	仿真激励文件, 用于对顶层文件sc_computer.v的仿真，已提供
|lpm_rom_irom.coe	测试程序的指令ROM初始值文件，已提供
|sc_datamem.coe	测试程序的数据RAM初始值文件，已提供

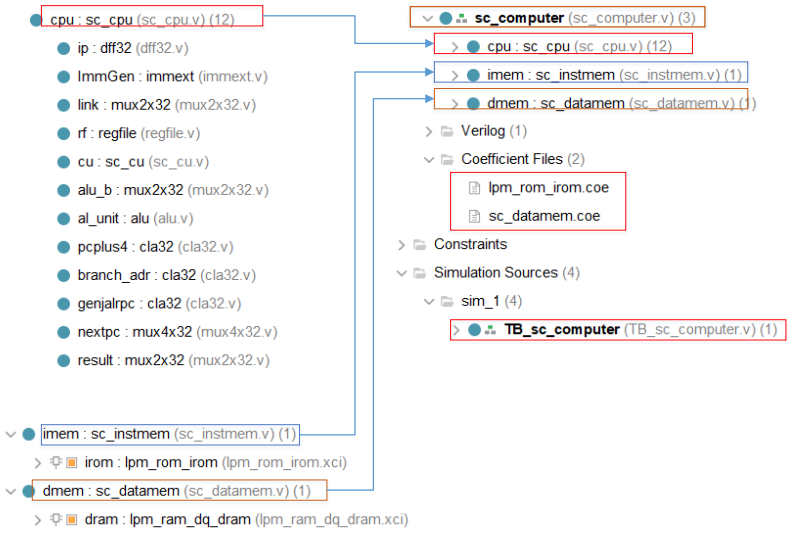


表 1: 实验文件树

2 实验原理

2.1 总体框架及通路图

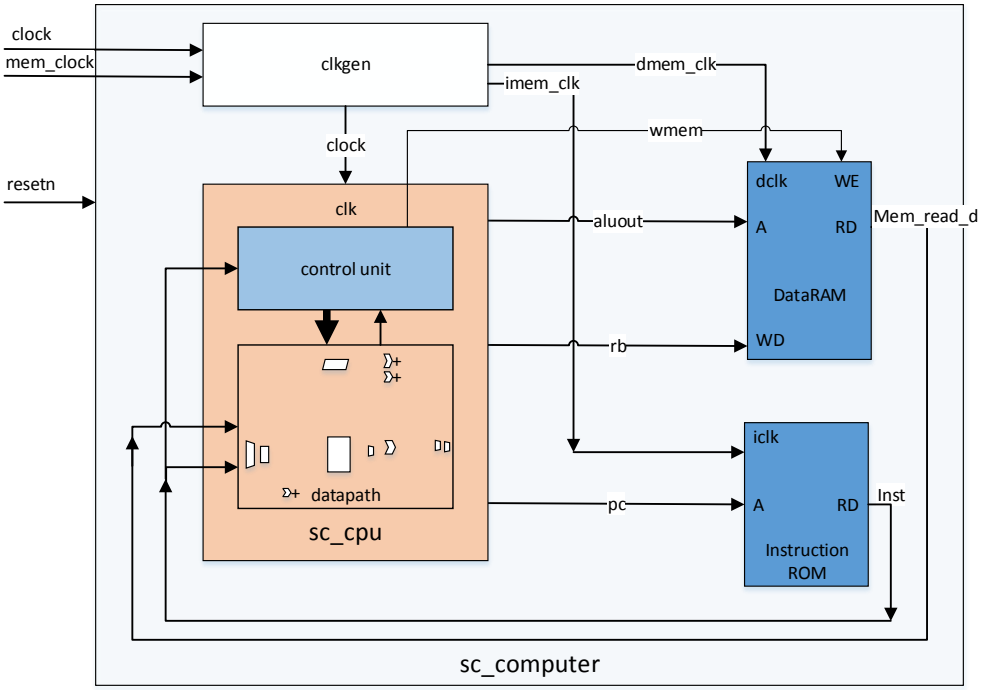


图 1: 单周期 CPU 框架图

如图 1，完整的单周期 CPU 实现框架图。其中datapath框外的部分和指令ROM已在实验二中完成，继续完成 datapath，即可将单周期 RISC-V软核完整实现。图中的clkgen模块目前是一个虚拟存在的模块，可根据第3部分里关于时钟的说明，自行在dataram和instructionROM模块里实现该部分对时钟的处理功能。

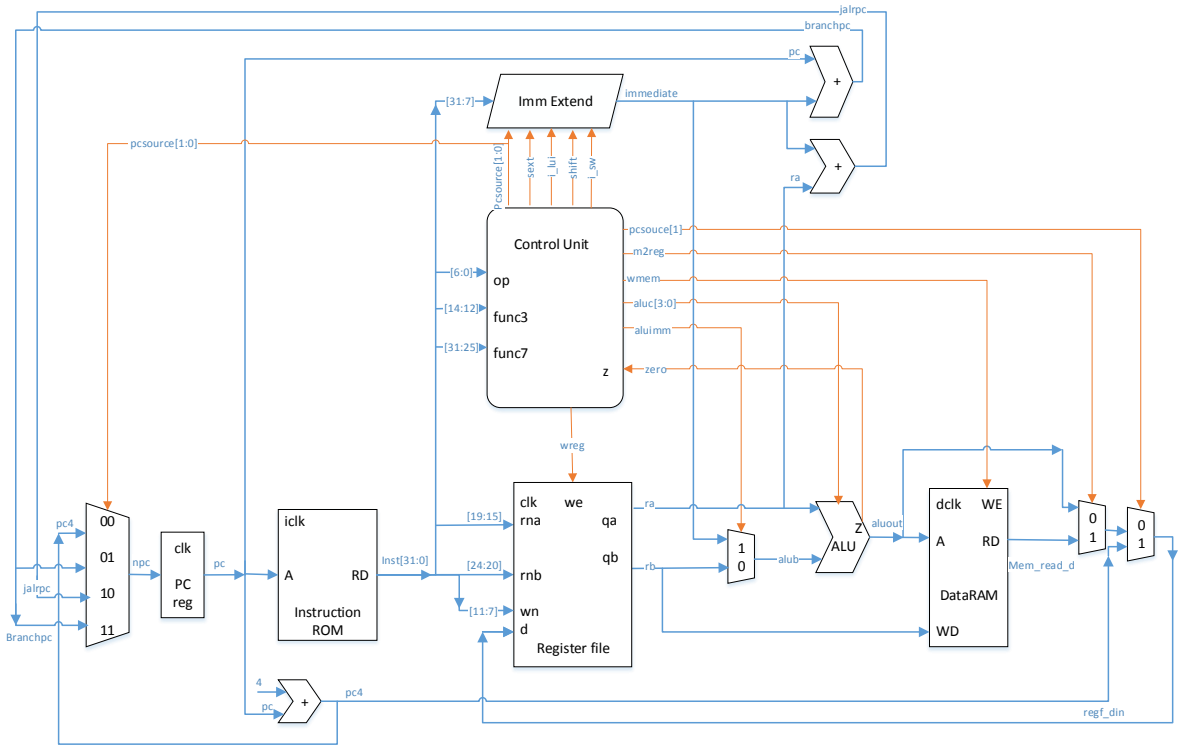


图 2: 支持22条指令的单周期 CPU 框架图

图2为支持22条指令的单周期CPU的完整通路图。实验三仅实现上述指令, 完整的RISC-V 指令集可于之后硬件综合设计中逐步完善，此处以掌握数据通路分析为主要目的。

2.2 控制器译码信号规范

2.2.1 ALU 控制码译码

$aluc_{3:0}$	功能	$aluc_{3:0}$	功能
0000	A + B(Unsigned)（add，addi，lw，sw，jalr）	0001	A<<B（sll，slli）
1000	A – B（sub，beq，bne）	0101	A>>B(logical)（srl，srli）
0111	A AND B（and，andi）	1101	A>>B(arithmetic)（sra，srai）
0110	A OR B（or，ori）	0010	B（lui）
0100	A XOR B（xor，xori）	其他	未使用（jal）

表 2: ALU 控制码译码信号表

参考实验一给出的一种码字定义，可如表2形式实现对这些指令的支持。这里需要注意，不按照表中的信号也可，只要保证不同类运算的aluc信号不同即可。

2.2.2 信号控制码译码信号

控制码译码信号与实验二相同，在通路连接时分别接入到需要控制的端口。参看如下表3并补充完整（同实验二表5）。

表 3: 控制信号译码

输入	alu 输出	sc_cn 输出									
指令类型	z	pcsource [1..0]	aluc [3..0]	aluimm	shift	sext	wmem	wreg	m2reg	i_lui	i_sw
add	x	0 0	0000	0	0	0	0	1	0	0	0
sub	x	0 0	1000	0	0	0	0	1	0	0	0
and			0111								
or			0110								
xor			0100								
sll	x	0 0	0001	0	1	0	0	1	0	0	0
srl			0101								
sra			1101								
addi	x	0 0	0000	1	0	1	0	1	0	0	0
andi	x	0 0	0111	1	0	1	0	1	0	0	0
ori											
xori											
slli	x	0 0	0001	1	1	0	0	1	0	0	0
srli											
srai											
lw	x	0 0	0000	1	0	1	0	1	1	0	0
jalr		1 0	0000								
sw											
beq	0	0 0	1000	0	0	1	0	0	0	0	0
	1	0 1(bpc)									
bne											
lui			0010								
jal	x	11(jpc)	xxxx	0	0	1	0	1	0	0	0

2.3 数据通路连接

在进行数据通路连接前，除了三大基本器件，还有些许小器件需要实现，包括加法器、触发器、多路选择器、移位器等。这些器件多为简单的组合逻辑，在数字逻辑课程已有涉及，此处不再赘述。

2.3.1 LW 指令

以 LW 指令为例构建基本的单周期 CPU 数据通路，需要的基本器件有PC、指令存储器、寄存器堆、数据存储器，见图 3，其他小的组合逻辑部件根据需求进行添加。

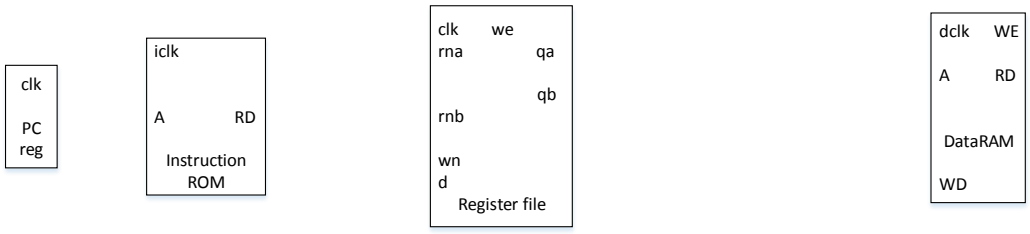


图 3

第一步为取指，将 PC(即指令地址) 输出至 Instruction Memory(指令存储器) 的 A 端口，如图 4。

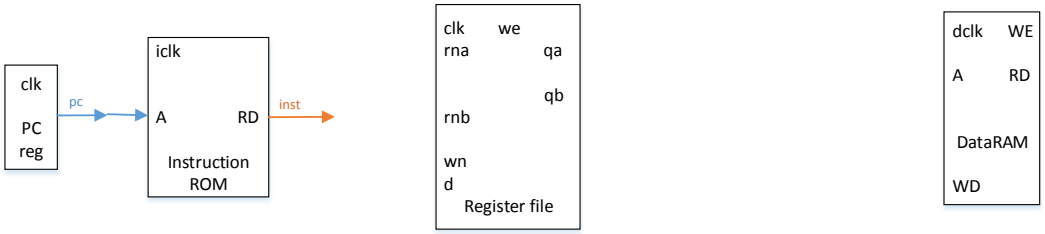


图 4

lw 指令的汇编格式为：lw rd,imm(rs1)，其中rs1(base基地址) 为指令[19:15] 所指向的寄存器值，imm(是地址偏移) 为指令 [31:20]。将rs1寄存器的值加上符号扩展后的地址偏移立即数immediate得到访存的地址，根据地址从存储器中读取 1 个字(连续 4 个字节) 的值写入到rd寄存器中。

根据lw 指令的定义，将指令存储器读出的指令 ([31:0]) 中 [19:15] 连接至寄存器堆的第一个输入地址rna。

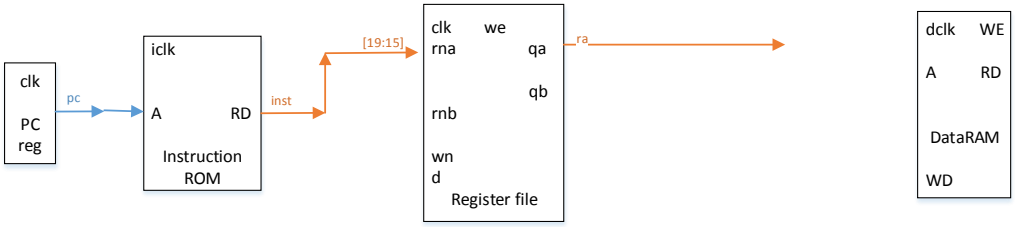


图 5

除此之外需要将imm进行扩展，因而将指令 [31:0] 传至有符号扩展模块immext，输出 32 位的符号扩展信号(immediate)。

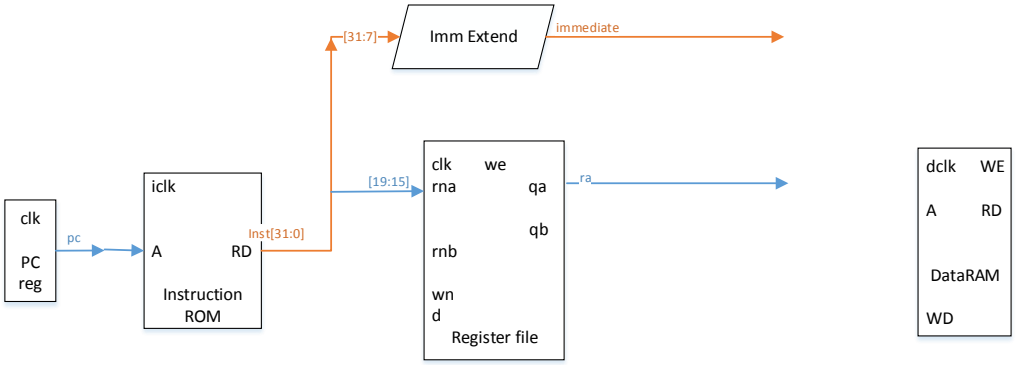


图 6

得到基地址 ra 和偏移量 immediate后，需进行加法计算，其操作可以采用如下描述：Addr = GPR[rs1]+ sign_extend(imm)。加法计算使用 ALU 中设计实现的加法运算，因而图7中，registerfile的qa口读出的ra 和经过有符号扩展后的sign_extend(imm)亦即immediate分别作为 ALU 的两个输入 (alua、alub)，经过 ALU 进行加法运算后，得到 aluout 作为地址，输入到数据存储器 Data RAM的 地址端口A。

这里需要注意的是，由于计算地址与进行加法运算相同，所以用于控制ALU运算类型的信号aluc 与加法运算应该相同，如图 7中红色部分所示。aluc 信号由sc_cu 译码得到，在此不再赘述。

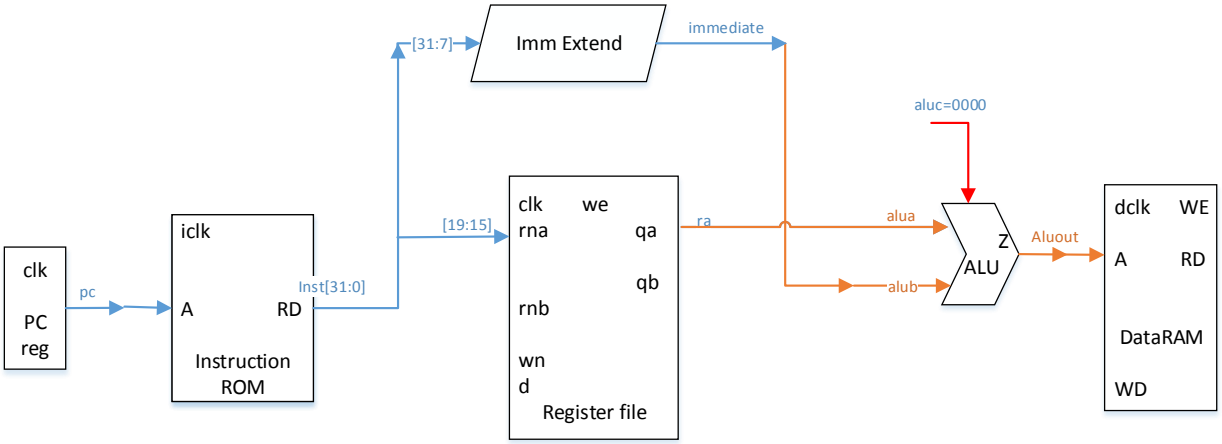


图 7

将地址输入后，将数据存储器读出的数据写回到 regfile 中，其地址为指令中的rd字段，即指令的[11:7]。如图 8，连接时需要将指令 [11:7] 连接至寄存器堆的wn端口，对应的数据信号mem_read_d连接至regfile的d 端口。

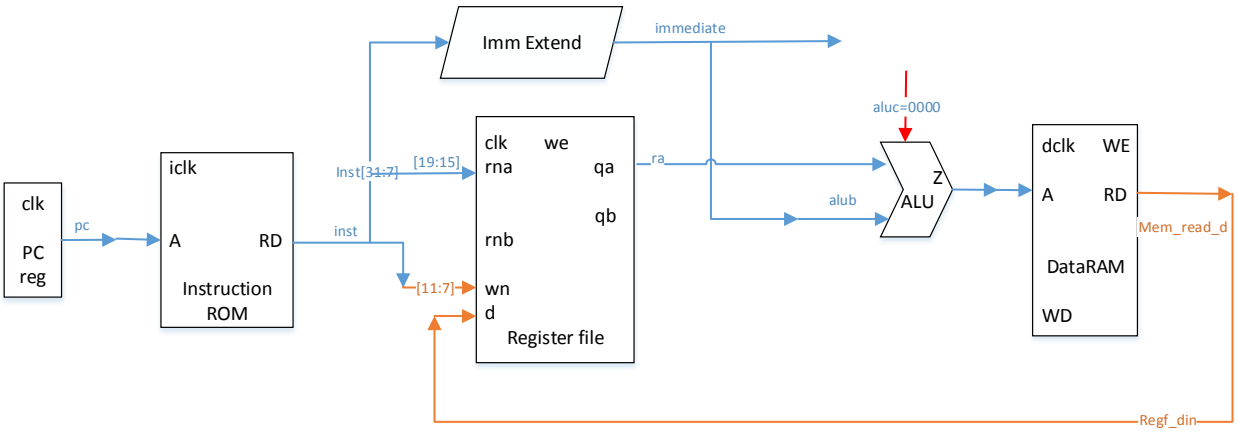


图 8

完成上述连接后，一条能够满足 lw 指令执行的数据通路即完成。lw 指令执行结束后，需开始下一条指令的执行，重复同样的执行过程。唯一的不同在于PC地址需要变为下一条指令所在地址。由于实现的是 32 位RISC-V 指令集，并且采用字节读写的方式，因为需要读取后续 4 个字节的数据，即PC+4。将得到的 PC+4 信号写入 PC(D 触发器) 的输入端，即可实现每周期地址 +4 的操作。

注意：这里我们不采用很多参考书籍中复用ALU的方式计算加法，而是使用一个单独的加法器模块，如图 9，原因在于，ALU 在实现多种运算逻辑后，其组合逻辑更加复杂，复用 ALU 进行 PC+4 的运算需要控制信号、输入的多路选择，增加了设计的复杂性的同时，也不符合硬件电路设计的思维。在电路设计中的复用更多是出于简化设计、优化延迟、降低功耗等目的，而非单一功能完成后进行复用的软件设计思维。

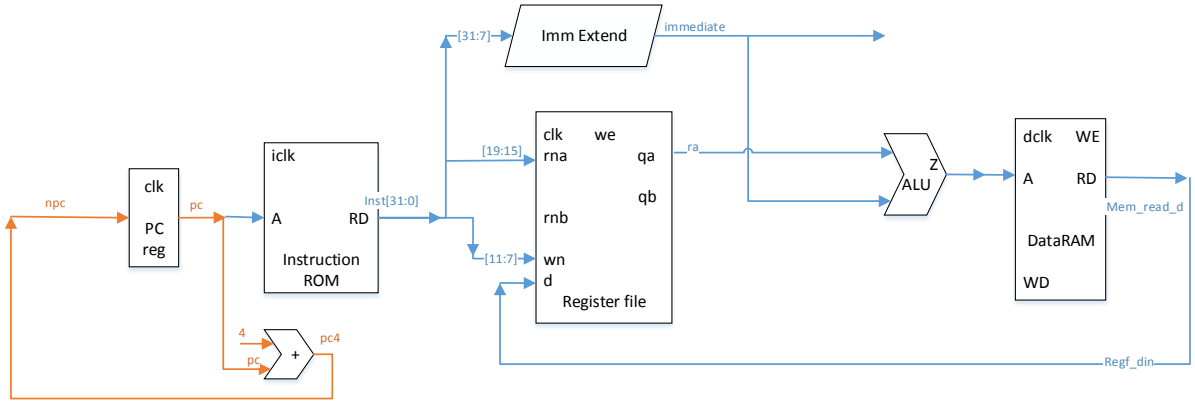


图 9

2.3.2 SW 指令

在完成 lw 指令的数据通路后，sw 指令仅需进行些许改动即可。首先来看 sw 指令的定义：sw rs2,imm(rs1)。从指令格式和译码的数据域来看，与 lw 无异。其区别在于，lw 需要从数据存储器读取数据并写入 regfile，而sw仅需要将数据写入数据存储器。计算地址的Addr = GPR[rs1]+sign_extend(imm)不变，但 rs2 变为读取寄存器，因而需要将其连接至 rnb，读出的数据RD2 信号连接至数据存储器的WD 端口。

这里涉及到几个红色标出的控制信号的值，分别为 wmem和 wreg，均连接至写使能端口 WE，值为 1 时使能，为 0 时不使能。如图 10所示，在进行存储器读写时，wmem需要使能，置 1。虽然此时执行的是 sw 写操作，但是 ReadData 信号会是什么值？这取决于使用的 Memory 类型，假设会有随机值读出，wn 同时将指令的 [11:7] 写入，这时将会有错误的值写到寄存器堆中。为了避免这种情况，需要将 wreg 置为 0，此时无法对寄存器堆进行写操作。这里进行延伸，所有不对寄存器堆进行写操作的指令，都可能会有类似的情况发生，因此这种类型的指令都应该将 regfile 的写使能关闭。同理，所有不对指令寄存器进行读写的指令，也应当关闭其使能端口，避免意外情况的出现。

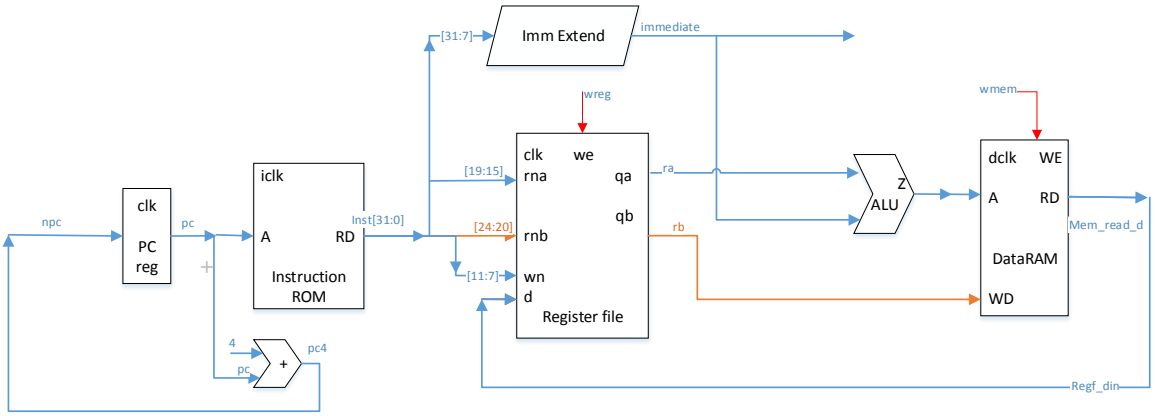


图 10

2.3.3 R-Type和I-Type 指令

R-Type 是RISC-V 指令中一大类指令的合集，均为 XXX rd,rs1,rs2 形式的指令。除了指令进行的运算类型不同外，其操作均为将 rs1,rs2 所在寄存器的值进行相应运算后，存入 rd 中。I-type指令与R-Type类似，为 XXX rd,rs1,imm 形式的指令，rs2 替换为立即数。对已经满足lw、sw 的数据通路进行一定改造即可。

通路改造方法为添加多路选择器。对于R-type指令，ALU 输入为ra,rb,(指令的rs1,rs2)，分别对应alua，alub。对于I-type指令，ALU 输入为ra,immediate,(指令的rs1和扩展后imm)，因此需要在alub处加入多路选择器，选择来源为寄存器堆registerfile的qb输出口信号rb还是符号扩展后的立即数immediate，控制信号为aluimm。最后写回到 regfile 的值应该为 ALU 计算得到的值，为aluout，加入多路选择器控制regf_din 来源为ALU 或数据存储器，控制信号为 m2reg(Memory to regfile)。

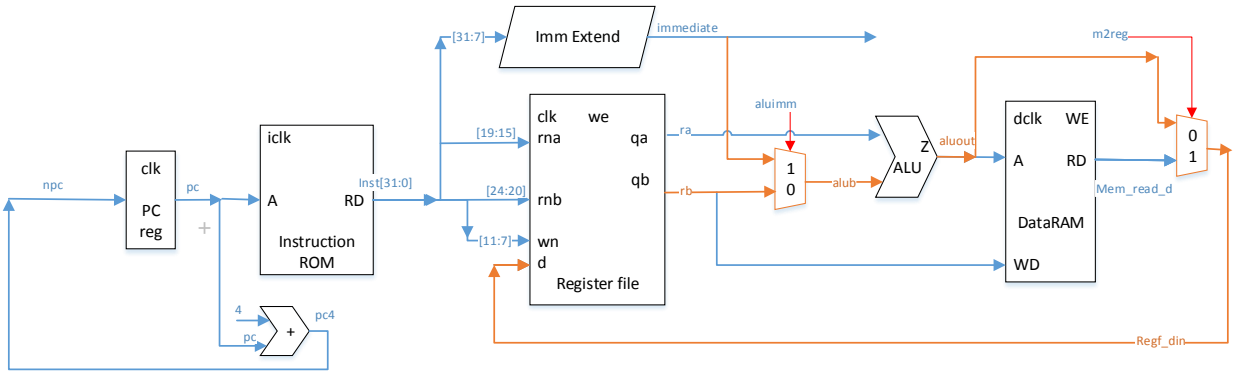


图 11

2.3.4 Branch 指令

接下来添加分支指令beq、bne，即通过条件判断是否需要跳转。此处以 beq 指令为例。其定义为：beq rs1, rs2, imm。如果寄存器 rs1 的值等于寄存器 rs2 的值则转移，否则顺序执行。转移目标地址由立即数imm 左移 1 位并进行有符号扩展的值加上该分支指令对应的指令PC指针计算得到。

注意：RISC-V指令架构中定义分支指令里的imm是单位为半字（2个字节）的地址偏移，因此需要在进行偏移量有符号扩展之后，左移一位，构成字节地址与PC相加。这步操作直接在立即数扩展模块immext内完成即可。

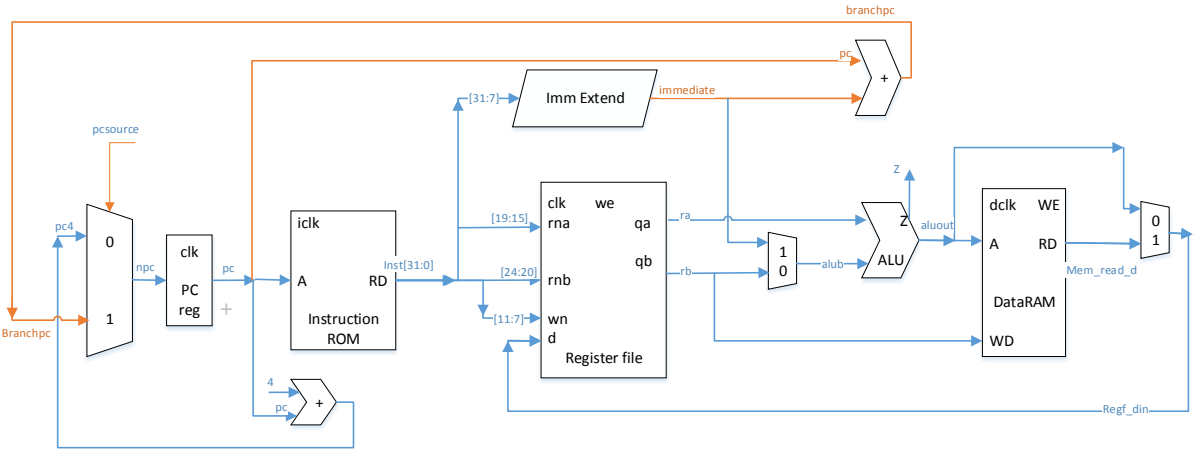


图 12

beq 需要三步操作，分别为条件判断，偏移计算，PC 转移，下面分别实现每步操作。条件判断需要判断 rs1、rs2 所在的寄存器值是否相等，可以使用 ALU 的减法进行计算，输出z信号，作为标志。之后在sc_cu模块内（图中略去该模块），与译码得到的 i_beq, i_bne 信号（判断是否为beq或bne分支指令）进行 and 操作，得到pcsource信号（pcsource = (i_beq & z) | (i_bne & ~z)）。

第二步偏移计算公式为：

- $target\_offset = Sign\_extend(imm||0^1)$
- $PC \leftarrow PC + target\_offset$

先将 imm 左移 1 位后，再进行有符号扩展，最后与 PC 相加。最后 PC 转移根据pcsource 信号进行控制，满足条件时，pcsource 为 1，选择 branchpc 作为下一条指令的地址。通路图见图 12。

2.3.5 jal和jalr 指令

jal跳转指令实际为无条件跳转，其格式为jal rd，imm，不需要做任何判断，因此更加简单。此时，需要有两步操作，其一是将当前指令的下一条指令之地址指针PC+4存入rd寄存器（该值即为函数调用时的返回地址指针）。其二，计算jalpc地址，更改 PC 。其跳转目标jalpc可以沿用上一节的branchpc信号，由该指令对应指令PC 与immediate相加得到。因为只要在immext模块里，选择此指令时输出的是经左移一位处理的jal指令符号扩展后偏移量值immediate即可（参见实验二的立即数模块设计指导部分）。

jalr指令是跳转后返回指令，其格式为jalr rd, imm(rs1)。此时，需要有两步操作，第一步是将当前指令的下一条指令之地址指针PC+4存入rd寄存器；第二步，将PC更改为新的跳转地址，其值来自于rs1寄存器的值ra（基地址）和立即数immediate（偏移量）的和。

此时，原来的二选一多路选择器需要做调整，改为4选1，控制信号pcsource也变为2bit，下一条PC的值npc由pcsource根据指令类型选定。具体可以定义如下：

pcsource[1:0]	instruction	npc
0 0	other inst	pc4
0 1	beq, ben	branchpc
1 0	jalr	jalrpc
1 1	jal	jalpc

```
pcsource[1] = i_jalr | i_jal;
pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_jal ;
```

如图 13，而后通过多路选择器。多路选择器直接由 pcsource[1:0]进行控制。

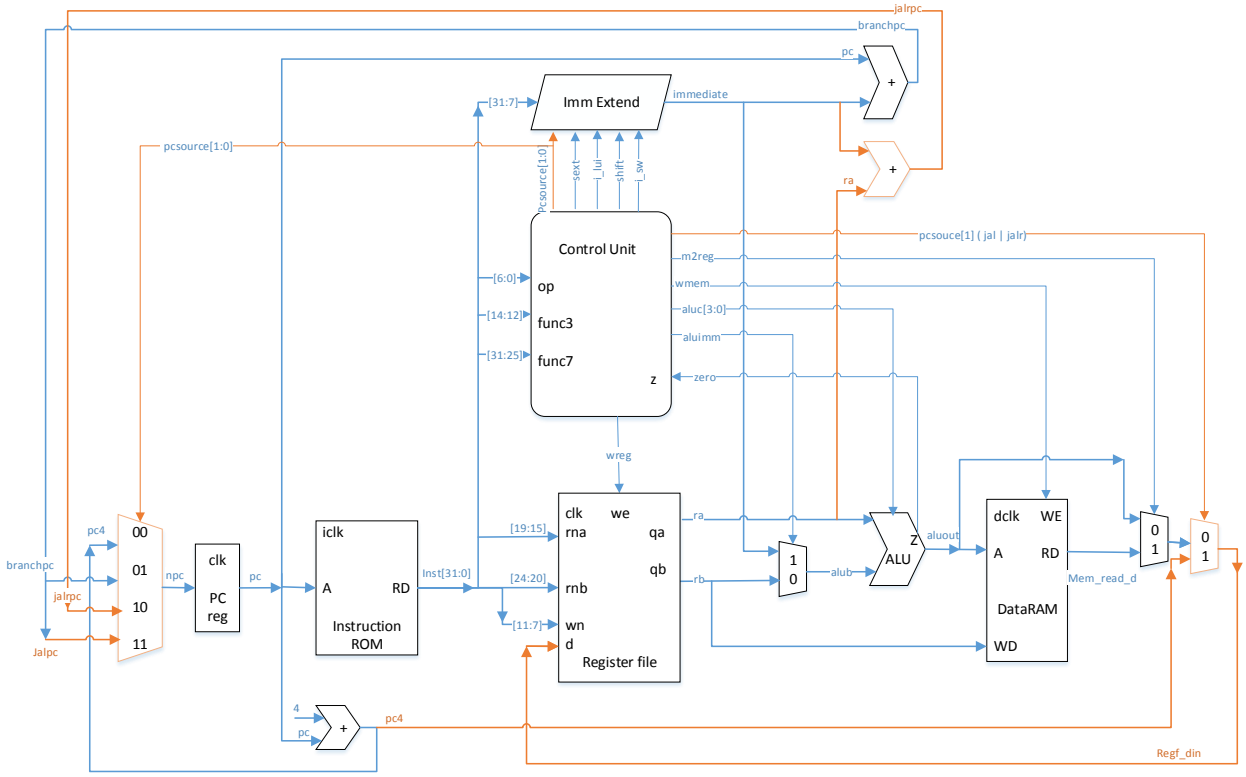


图 13

2.3.6 lui 指令

lui指令格式为lui rd,imm，请根据其指令功能和定义，自行分析要支持该指令，是否要在图13基础上添加新的连接，需要修改什么部分？对照下图中绿色和黄色连线的部分思考。

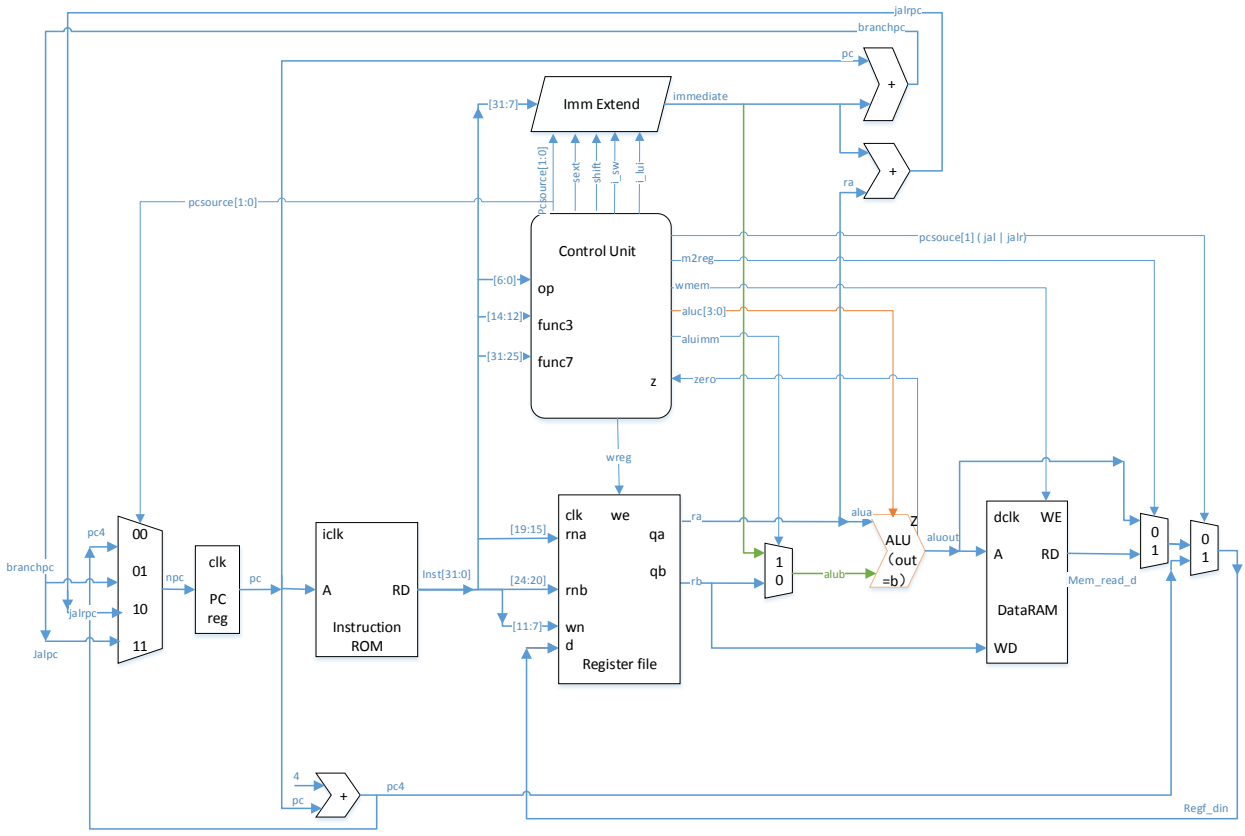


图14

3 仿真验证

3.1 测试程序

CPU运行的汇编代码沿用实验二中lab2.2产生的机器码，用以验证本CPU设计对22条指令的译码执行过程的支持。该段示例的CPU程序代码，涵盖了所有应实现待验证的RISC-V指令类型。其功能包括完成四个数的求和运算、数据的逻辑及算术移位操作运算、函数调用及返回、有条件和无条件跳转等功能。

3.2 仿真激励文件Testbench与仿真验证

测试用仿真激励文件TB_sc_computer.v随实验包提供。在Vivado环境内用工具内已含的仿真工具Modelsim完成功能仿真，对照仿真波形，检查指令的执行结果与实验二中的RISC-V CPU模型虚拟仿真结果是否一致。通过仿真波形，查看示例程序的运行过程是否符合程序设想，PC指针、指令数据、结果数据等是否正确。

图15（a）为仿真结果波形总图，程序执行到最后在1515.1ns，停在最后一条死循环跳转指令处，此时寄存器Reg18的值为0001ffff，指令inst=0000006f，PC指针pc=0000007c。

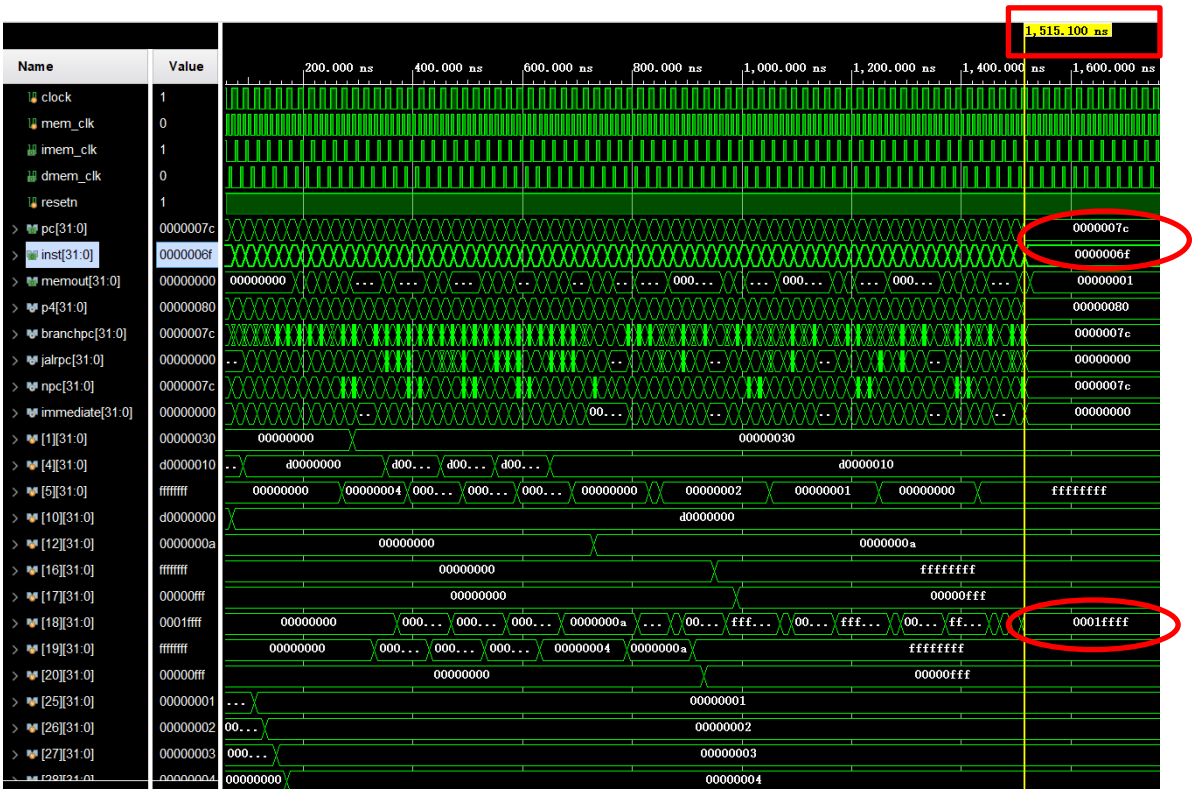


图15（a）

图15（b）中标注了程序执行到790ns处的情形。这时可以看到寄存器reg18的值为0000000a。是四个数1，2，3，4之和。此刻执行的指令是 lw x19，0(x4)，从存储器中读到的0000000a被正确地读入寄存器reg19中，reg19的值改变为0000000a。说明前面的调用子程序循环计算的一系列指令得到正确执行，计算结果被正确写入数据存储器，并在此时被正确读出来写入寄存器reg19。读写数据存储器的指令也得到正确执行。

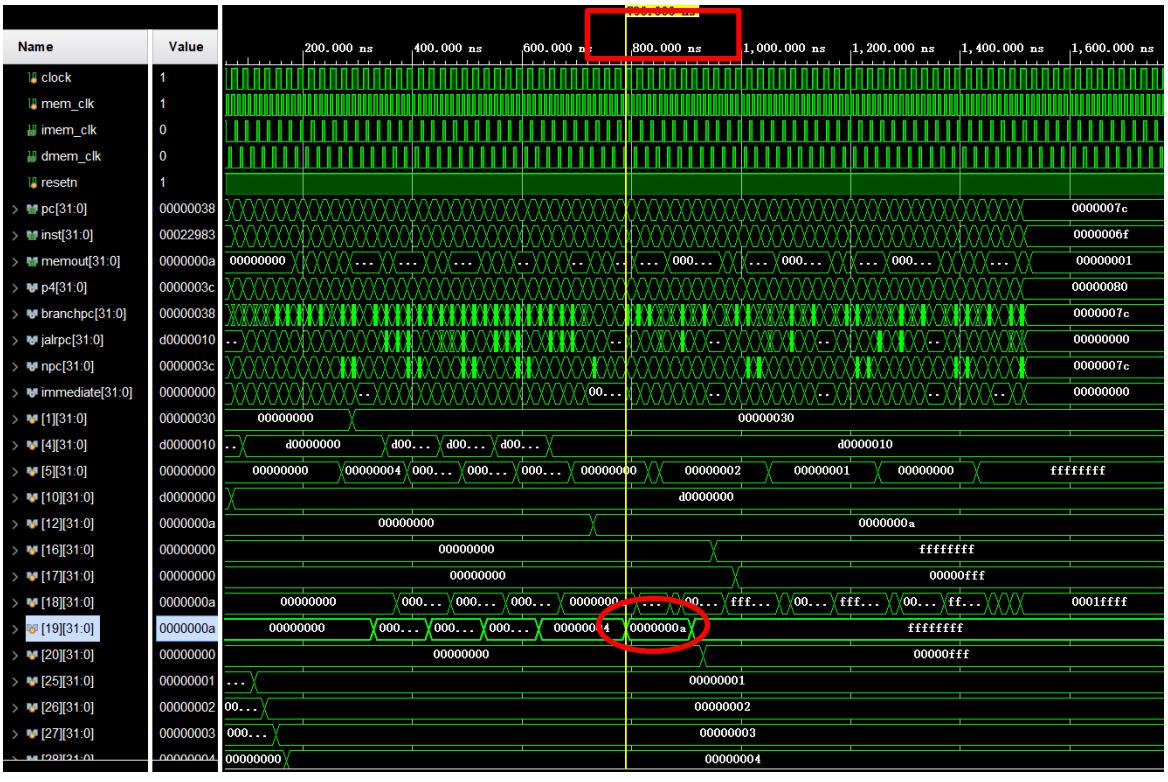


图15（b）

3.3 关于单周期 CPU 设计实验中的时钟问题

单周期 CPU 设计实验中除了一个基本的 clock 时钟信号外，为什么还设计有一个 2 倍频于 clock 信号、标记为 mem_clk 的时钟信号?

这是每个指令周期内，对各逻辑功能部件的工作时间顺序进行控制的需要。

从单周期 CPU 的设计原理上，如果指令 ROM 和数据 RAM 都为异步器件，即只要收到地址就开始送出数据，则在实现中只需要一个 clock 时钟信号控制 PC 的变化、以及整条指令的执行时间（机器周期）就够了。

但在EGO1实验板上 Xilinx 的 Artix-7系列 FPGA 器件中，只提供同步ROM 和同步 RAM 宏模块。同步 ROM 和同步 RAM， 其读写时序如图 16 和图 17 所示。

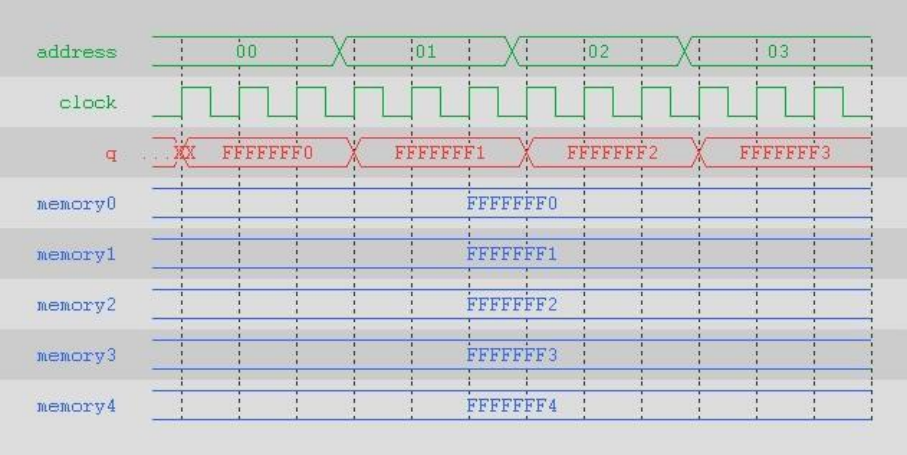


图 16. 同步 ROM 和同步 RAM 的“读”操作工作时序

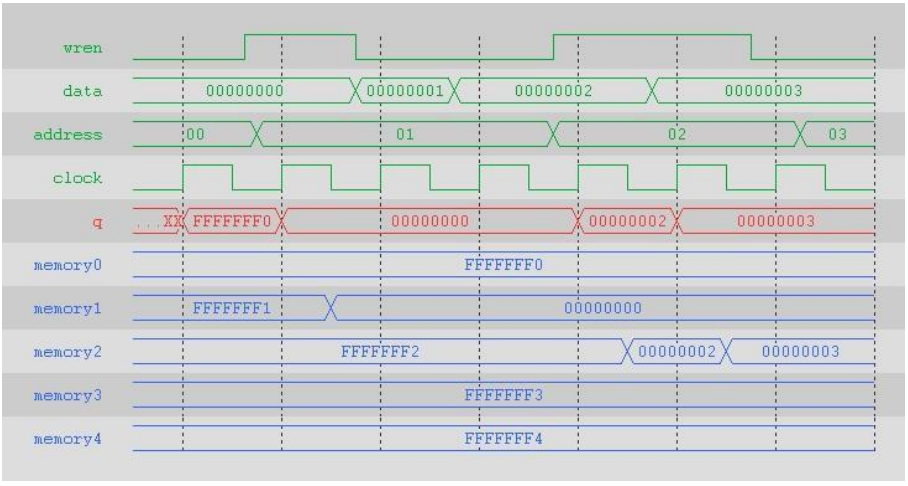


图 17. 同步 RAM 的“写”操作工作时序

考察图 16所示同步读时序，当同步 ROM 或 RAM 接收到数据地址时，不同于异步器件在经过一段组合电路的反应延迟后送出数据，同步器件需要等到同步时钟信号 clock 上升沿时才真正送出数据（这就可以使延迟时间不尽相同的多个器件在输出级达成一致）。考察图 17 所示同步写时序，同步 RAM 在接收到地址后，在 wren 信号高有效期间，需要等到同步时钟 clock 上升沿，才开始写动作，并将要写入的新数据马上输出到 q 输出端，即在写时可以同时读输出，满足图 16 的读时序，并在同步时钟 clock 的下降沿将数据真正写入到存储器中。

对于单周期 CPU 指令执行的一系列过程（5 个大步骤）：

① 取指令：PC 在时钟上沿改变新值，按改变后并稳定的新 PC 值作为指令地址， 访问指令 ROM 取出指令；

② 译码：指令的一部分交由控制器 CU 进行译码，另一部分至寄存器堆取操作数；

③ 执行：ALU 等部件执行相应操作；

④ 访问数据 MEM： 读或写数据存储器；

⑤ 回写寄存器：回写结果至目的寄存器。

以上各步骤在逻辑上是顺序执行的，在时间上是有先后顺序的，因此，对于采用有同步信号的指令同步 ROM、数据同步 RAM，在一个 CPU 指令周期内，需要安排更精细的时钟信号来协调它们的顺序工作过程。

为此，实验中设计了 2 路时钟信号，一路为表征机器周期的 clock 时钟信号，另一路为 clock 时钟信号 2 倍频的 mem_clk 时钟信号。事实上，在实现中，clock 是mem_clk 信号的 2 分频信号，分频比倍频在电路上要容易实现的多。信号波形如图18所示。

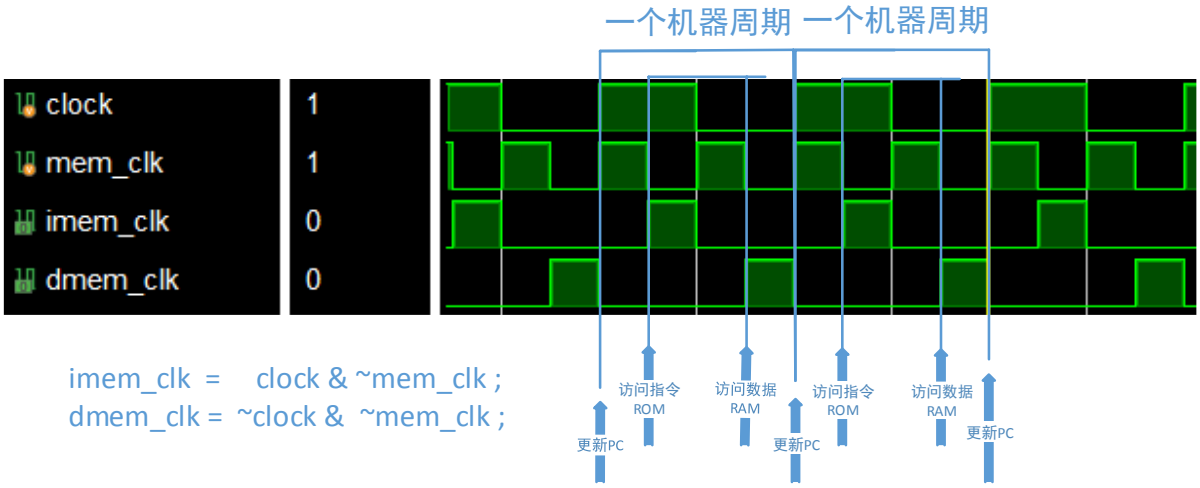


图 18. 一个机器周期内各执行步骤的执行触发顺序

如图18所示，由输入信号 clock 和 mem_clk 可通过组合逻辑生成 4 个节拍信号，实验中根据时序需要，选取图中标注为 imem_clk、dmem_clk 的两个节拍分别作为指令 ROM 和数据 RAM 的同步时钟。这样，CPU 的工作就在一个机器周期内，有条不紊地顺序执行。各步骤上升沿间的时间间隔，就是留给各阶段组合逻辑的信号传输及延迟时间。

利用机器周期和更高频率（如 8 倍频）信号的组合，可以将一个过程分为更多的节拍（如 8 个节拍），从而安排更多功能部件的协调有序执行。

4 拓展思考

对比Ripes仿真软件里的32bit单周期RISC-V架构CPU框架图，可见本实验中完成的数据通路中多用了两个加法器用于计算分支和跳转指令的转移地址。如果考虑尽量减少使用硬件资源，用于产生branchpc 和jalrpc 地址的加法器用ALU来计算目标地址，设计中应该有哪些修改？修改后整体电路在FPGA资源的使用上与之前相比有什么变化？

5 实验报告要求

实验汇报应简明扼要，重点突出。具体内容包括以下几部分：

1. 实验目的。
2. 本次实验所用的实验平台、仪器以及其它实验器材和部件等。
3. 实验任务。这部分是实验报告最主要的内容。根据任务要求，描述设计原理与思路，设计的实现方式，结构框架等。（不要仅贴整段Verilog 程序代码，而是以表述介绍设计思路为主，如有必要，可贴关键代码段，并加注释）。结合实验原理，描述设计的全过程和实验步骤。
4. 实验结果演示总结。提供仿真结果整体波形和关键点截图及解释，不要只贴图。可参看对图15解释的语句，指出对应该部分截图要说明的关键信息，如该时刻pc值、inst指令值、该指令执行后什么单元会受影响，哪个寄存器或存储器的值变成了什么。是否跟你之前完成的lab2.1中CPU模型仿真的表1结果一致？
5. 对拓展思考问题的理解与实践（选做）。

A RISC-V-Reference-Data

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

①

RISC-V Reference Data

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'bM[](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'bM[](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] = {32'bimm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](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 regsiters
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 regsiter
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
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 upper Half	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
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)
fmnsub.s,fmnsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fmnadd.s,fmnadd.d	R	Negative Multiply-ADD	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 from DP to SP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert from SP to DP	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)

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

PSEUDO INSTRUCTIONS

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

REGISTER NAME, USE, CALLING CONVENTION

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

OPCODES IN NUMERICAL ORDER BY OPCODE ③

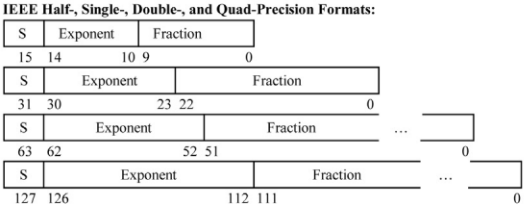
MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	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	I	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

IEEE 754 FLOATING-POINT STANDARD ④

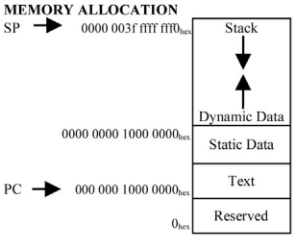
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

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

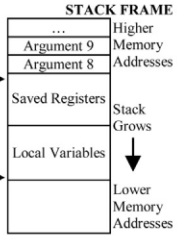
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



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