SI100B EE-Part MiniCPU

RISC-V Assembly

Yuxuan Li

VSPLab@SIST

2026.05.28

liyx22025@shanghaitech.edu.cn

目录

1 指令的操作数：寄存器与立即数

2 算术指令

3 读写指令

4 跳转指令

5 总结

1 指令的操作数：寄存器与立即数

指令的操作数

汇编指令有两类可能的操作数：寄存器和立即数

- 指令 `add x3, x1, x2` 的含义是：将 `x1` 和 `x2` 中存储的数相加，写入 `x3` 中。
- 指令 `addi x3, x1, 4` 的含义是：将 `x1` 中存储的数加上 `4` 后，写入 `x3` 中。
- 用数学公式表达：$x_3 = x_1 + x_2$、$x_3 = x_1 + 4$
- 立即数：上述的 `4` 就是立即数，即指令中“字面意义的数”。
- 寄存器：上述的 `x1` 和 `x2` 就是寄存器。

寄存器堆

汇编指令无法直接访问内存！因为内存的读写速度远远慢于处理器执行的速度。

在处理器中，有一个称为寄存器堆（Register File）的结构，在 RISC-V 架构中，寄存器堆由 32 个寄存器组成，这些寄存器的访问速度非常快，可以被指令直接访问。

- 寄存器依次被记为 x0、x1、x2、$\cdots$、x31。
- 寄存器可以用五位地址编码（$\log_2 32 = 5$），例如 x5 的地址是 00101。
- 寄存器的宽度是 32 位的，这意味着每个寄存器都可以存储一个 32 位的整数。
- 寄存器 x0 是非常特殊的！它被硬编码为 0 且忽略所有写入。

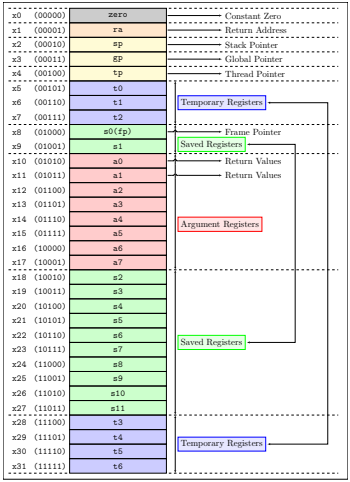
简单来说，寄存器堆可以视为一个存储模块，它总共可以保存 32 个 32 位数据。

寄存器堆

实际上，寄存器堆中的 32 个寄存器在使用上并不是完全随意的，而是被分配了特定的功能和名称

- 零寄存器：x0 也可记为 zero。
- 栈指针寄存器：x2 也可记为 sp。
- 保存寄存器：例如 x19 也可记为 s3，共 12 个。
- 临时寄存器：例如 x28 也可记为 t3，共 6 个。
- 参数寄存器：例如 x13 也可记为 a3，共 8 个。

不过，这些功能划分主要和函数调用有关，在本课程中并不涉及。因此，仍然可以当成 32 个相同的寄存器来使用，并继续使用 x0、x1、x2、⋯、x31 的名称。



寄存器堆

处理器在执行每一条指令时，对于寄存器堆存在以下操作

- 自寄存器堆中，读取至多两个数据（add 需要读两个，addi 需要读一个）。
- 向寄存器堆中，写入至多一个数据（部分指令不需要写寄存器）。

REGFILE

REGFILE 的端口设计如图1所示

- A1 是第一个读地址，RD1 是第一个读数据。
- A2 是第二个读地址，RD2 是第二个读数据。
- A3 是写地址，WD3 是写数据。
- WE 是写使能，只有在 WE 为高时才进行写入。[a]
- 在 add x3, x1, x2 中，A1 = 00001，A2 = 00010，A3 = 00011，WE = 1。RD1 和 RD2 是从 x1 和 x2 中读取的数据，WD3 是后续模块计算的 x1 + x2 的值。

[a]为什么不需要读使能呢?

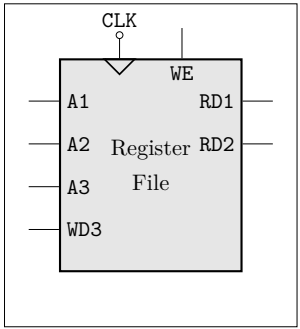


图 1: Register File

2 算术指令

加法指令 add

加法指令（Add）

代码 1: 加法指令

```
1 add x3, x1, x2                # x3 = x1 + x2
```

减法指令 sub

减法指令（Substract）

代码 2: 减法指令

```
1 sub x3, x1, x2                # x3 = x1 - x2
```

立即数加法指令 addi

立即数加法指令（Add Immediate）

代码 3: 立即数加法指令

```
1 addi x3, x1, 4          # x3 = x1 + 4
2 addi x3, x2, -8         # x3 = x2 - 8
```

请注意，只有 addi 而不存在 subi，这是因为立即数可以是负数！

立即数加法指令 addi

立即数加法指令有两个围绕 x0 的特殊用法

- 使用 addi x3, x0, 128，由于 x0 恒定为 0，相当于给 x3 直接填充 128。
- 使用 addi x0, x0, 0，由于任何写入 x0 的都会被忽略，这条指令相当于什么都不做的空指令 nop。由此可见，x0 可以作为丢弃计算结果的“黑洞”使用。

代码 4: 立即数加法的特殊用法

```
1 addi x3, x0, 128          # x3 = 128
2 addi x0, x0, 0            # nop
```

空指令 nop

实际上，你可以直接在汇编中使用 nop 代表什么都不做！

代码 5: 空指令

```
1 nop                          # pseudoinstruction
```

然而 nop 并不是一条真正的指令，而是一条伪指令（Pseudoinstruction）。伪指令是为了便利汇编语言而存在的，它会在汇编过程中转换为处理器真正支持的指令，例如 nop 对应的就是 addi x0，x0，0。实际上 RISC-V 未将 nop 作为真正的指令也凸显了其精简指令集的设计原则。在本课程中，不需要考虑任何伪指令的处理。

算术指令示例

代码 6: 算术指令示例

```
1 addi x1, x0, 9          # x1 = 9
2 addi x2, x0, 5          # x2 = 5
3 add x3, x1, x2          # x3 = x1 + x2
4 sub x4, x1, x2          # x4 = x1 - x2
5 addi x5, x4, -5         # x5 = x4 - 5
```

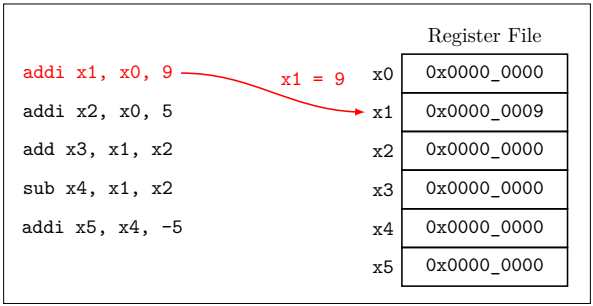
算术指令示例：初始状态

```
addi x1, x0, 9
addi x2, x0, 5
add x3, x1, x2
sub x4, x1, x2
addi x5, x4, -5
```

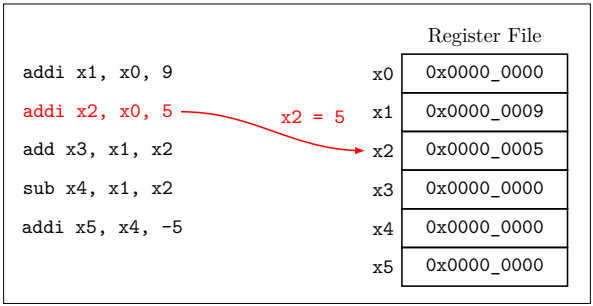
Register File

x0	0x0000_0000
x1	0x0000_0000
x2	0x0000_0000
x3	0x0000_0000
x4	0x0000_0000
x5	0x0000_0000

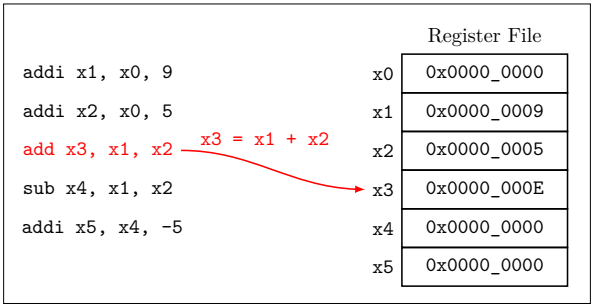
算术指令示例：周期 1


addi x1, x0, 9
addi x2, x0, 5
add x3, x1, x2
sub x4, x1, x2
addi x5, x4, -5
x1 = 9
Register File
x0 0x0000_0000
x1 0x0000_0009
x2 0x0000_0000
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0000

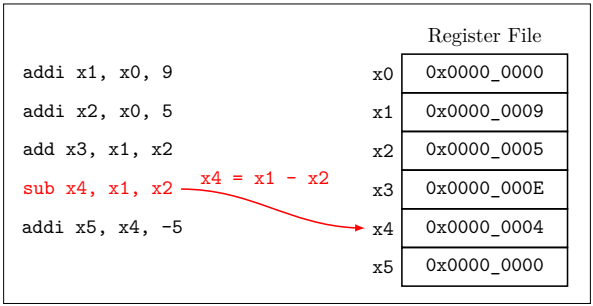
算术指令示例：周期 2


Register File
addi x1, x0, 9
addi x2, x0, 5
add x3, x1, x2
sub x4, x1, x2
addi x5, x4, -5
x2 = 5
x0 0x0000_0000
x1 0x0000_0009
x2 0x0000_0005
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0000

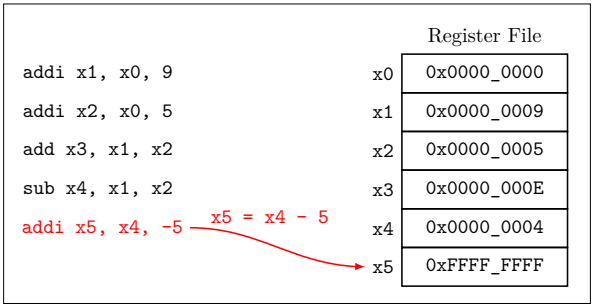
算术指令示例：周期 3


addi x1, x0, 9
addi x2, x0, 5
add x3, x1, x2
sub x4, x1, x2
addi x5, x4, -5
x3 = x1 + x2
Register File
x0 0x0000_0000
x1 0x0000_0009
x2 0x0000_0005
x3 0x0000_000E
x4 0x0000_0000
x5 0x0000_0000

算术指令示例：周期 4


addi x1, x0, 9
addi x2, x0, 5
add x3, x1, x2
sub x4, x1, x2
addi x5, x4, -5
x4 = x1 - x2
Register File
x0 0x0000_0000
x1 0x0000_0009
x2 0x0000_0005
x3 0x0000_000E
x4 0x0000_0004
x5 0x0000_0000

算术指令示例：周期 5


Register File
addi x1, x0, 9
addi x2, x0, 5
add x3, x1, x2
sub x4, x1, x2
addi x5, x4, -5
x5 = x4 - 5
x0 0x0000_0000
x1 0x0000_0009
x2 0x0000_0005
x3 0x0000_000E
x4 0x0000_0004
x5 0xFFFF_FFFF

位运算

位运算：对于两个数的每一个二进制位进行逻辑运算

- 按位与：`3 & 5 = 0011 & 0101 = 0001 = 1`
- 按位或：`3 | 5 = 0011 | 0101 = 0111 = 7`

按位与指令 and andi

按位与指令（And）

代码 7: 按位与指令

```
1 and x3, x1, x2              # x3 = x1 & x2
2 andi x3, x1, 255            # x3 = x1 & 0x0000_00FF
```

与运算：两个输入同时都为 1 时，输出为 1。

按位或指令 or ori

按位或指令（Or）

代码 8: 按位或指令

```
1 or x3, x1, x2                 # x3 = x1 | x2
2 ori x3, x1, 255               # x3 = x1 | 0x0000_00FF
```

或运算：两个输入有一个为 1 时，输出为 1。

当小于时赋值指令 slt slti

当小于时赋值指令（Set Less Than）

代码 9: 当小于时赋值指令

```
1 slt x3, x1, x2          # x3 = (x1 < x2) ? 1 : 0
2 slti x3, x1, 32         # x3 = (x1 < 32) ? 1 : 0
```

若小于，则赋值为 1，否则赋值为 0。

3 读写指令

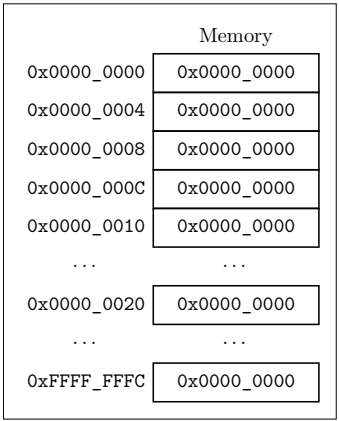
内存读写

为什么处理器需要进行内存读写？

- 寄存器堆总共只能存储 32 个 32 位数，需要更多空间存放程序的中间结果。
- 内存是一片更大的存储空间，可以存储很多的 32 位数。
- 内存无法被大部分指令直接访问，但是有两类特殊的读写指令可以和内存交互。
- 读指令可以将一个内存中的数据搬运到寄存器中（读：处理器从内存读）。
- 写指令可以将一个寄存器中的数据搬运到内存中（写：处理器向内存写）。

内存读写

- 内存地址是按 4 递增的，原因是：内存中存储的数据是 32 位的，但地址按字节记。
- 32 位系统：地址和数据均用 32 位表示。
- 32 位系统的最大寻址范围是 4 GB。
- 读指令的目标：`x1 = mem[0x10]`。
- 写指令的目标：`mem[0x20] = x2`。



读指令 lw

读指令（Load Word）

代码 10: 读指令

```
1 addi x5, x0, 16              # x5 = 0x10
2 lw x1, 0(x5)                 # x1 = mem[0x10]  0(x5) -> x5 + 0 = 0x10
3 lw x1, 4(x5)                 # x1 = mem[0x14]  4(x5) -> x5 + 4 = 0x14
```

写指令 sw

写指令（Store Word）

代码 11: 写指令

```
1 addi x5, x0, 32              # x5 = 0x20
2 sw x2, 0(x5)                 # mem[0x20] = x2  0(x5) -> x5 + 0 = 0x20
3 sw x2, 4(x5)                 # mem[0x24] = x2  4(x5) -> x5 + 4 = 0x24
```

读写指令与立即数偏移

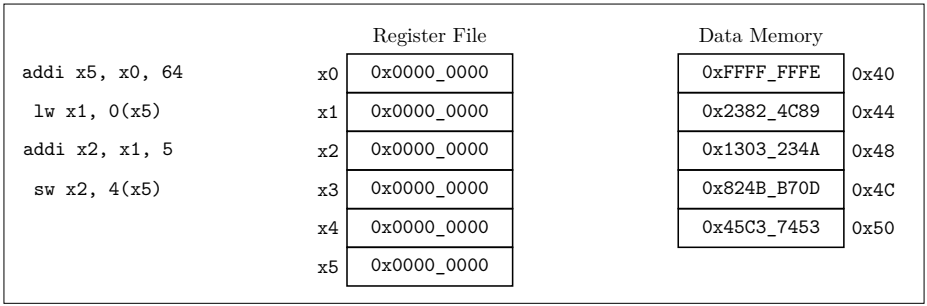
- 读写指令中都使用了类似 `0(x5)` 或 `4(x5)` 的记号，这实际上是一种立即数偏移的语法，代表对 `x5 + 0` 或 `x5 + 4` 地址处的内存空间进行读写操作。
- 读写指令采用这种设计的原因是：指令的长度是有限的，无法将地址本身编码到指令中，因此需要先将目标地址加载到某个寄存器中，例如 `x5`。如果接下来访问的是该地址后的一段连续空间，立即数偏移的语法可以避免重复加载地址。
- 此处的立即数和 `addi` 中的立即数没有本质性区别，只是写法不同。
- 由此可见，读写指令在执行时同样需要进行一次加法计算！

读写指令示例

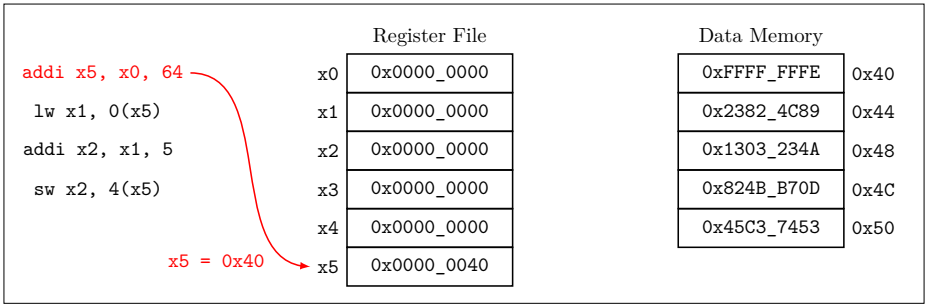
代码 12: 读写指令示例

```
1 addi x5, x0, 64              # x5 = 0x40
2 lw x1, 0(x5)                 # x1 = mem[0x40]  0(x5) -> x5 + 0 = 0x40
3 addi x2, x1, 5               # x2 = x1 + 5
4 sw x2, 4(x5)                 # mem[0x44] = x2  4(x5) -> x5 + 4 = 0x44
```

读写指令示例：初始状态


addi x5, x0, 64
lw x1, 0(x5)
addi x2, x1, 5
sw x2, 4(x5)
Register File
x0 0x0000_0000
x1 0x0000_0000
x2 0x0000_0000
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0000
Data Memory
0xFFFF_FFFE 0x40
0x2382_4C89 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50

读写指令示例：周期 1


addi x5, x0, 64
lw x1, 0(x5)
addi x2, x1, 5
sw x2, 4(x5)
x5 = 0x40
Register File
x0 0x0000_0000
x1 0x0000_0000
x2 0x0000_0000
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0040
Data Memory
0xFFFF_FFFE 0x40
0x2382_4C89 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50

读写指令示例：周期 2


addi x5, x0, 64
lw x1, 0(x5)
addi x2, x1, 5
sw x2, 4(x5)
Register File
x0 0x0000_0000
x1 0xFFFF_FFFE
x2 0x0000_0000
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0040
x1 = mem[0x40]
Data Memory
0xFFFF_FFFE 0x40
0x2382_4C89 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50

读写指令示例：周期 3


addi x5, x0, 64
lw x1, 0(x5)
addi x2, x1, 5
sw x2, 4(x5)
x2 = x1 + 5
Register File
x0 0x0000_0000
x1 0xFFFF_FFFE
x2 0x0000_0003
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0040
Data Memory
0xFFFF_FFFE 0x40
0x2382_4C89 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50

读写指令示例：周期 4


addi x5, x0, 64
lw x1, 0(x5)
addi x2, x1, 5
sw x2, 4(x5)
Register File
x0 0x0000_0000
x1 0xFFFF_FFFE
x2 0x0000_0003
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0040
mem[0x44] = x2
Data Memory
0xFFFF_FFFE 0x40
0x0000_0003 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50

DMEM

DMEM 的端口设计如图2所示

- A 是地址，读写地址均从这里传入。
- RD 是读出的数据。
- WD 是写入的数据。
- WE 是写使能，在 WE 为高时写，在 WE 为低时读。
- 读写不会同时发生！因此只需要一个地址端口就够了。

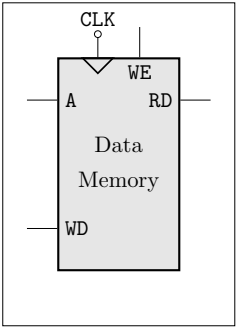


图 2: Data Memory

4 跳转指令

分支

程序并不总是顺序执行，往往需要这样的结构

- 判断：当满足条件时执行一些指令，当不满足条件时执行另一些指令。
- 循环：重复执行一些指令，直到满足某种条件。

在汇编层面，这些控制流的实现，都需要依靠分支指令进行实现。

分支指令 bne

分支指令（Branch if Not Equal）

代码 13: 分支指令

```
1 addi x1, x0, 8              # x1 = 8
2 addi x2, x0, 4              # x2 = 4
3 bne x1, x2, target          # if x1 != x2: -> target
4 addi x3, x0, 73             # x3 = 73 (skip)
5 target:
6 addi x4, x0, 74             # x4 = 74
```

分支指令 bne

RISC-V 支持许多不同的分支指令，但本课只要求实现 bne

- `beq x1, x2, target`，若相等则跳转（Branch if Equal）。
- `bne x1, x2, target`，若不等则跳转（Branch if Not Equal）。
- `blt x1, x2, target`，若小于则跳转（Branch if Less）。
- `bge x1, x2, target`，若大于等于则跳转（Branch if Greater or Equal）。
- `bltu x1, x2, target`，这是 `blt` 的无符号数版本。
- `bgeu x1, x2, target`，这是 `bge` 的无符号数版本。

思考：为什么指令集设计上只支持了“小于”和“大于等于”？

控制流的实现：存储指令！

然而，处理器在遇到分支指令时，是如何完成跳转的呢？

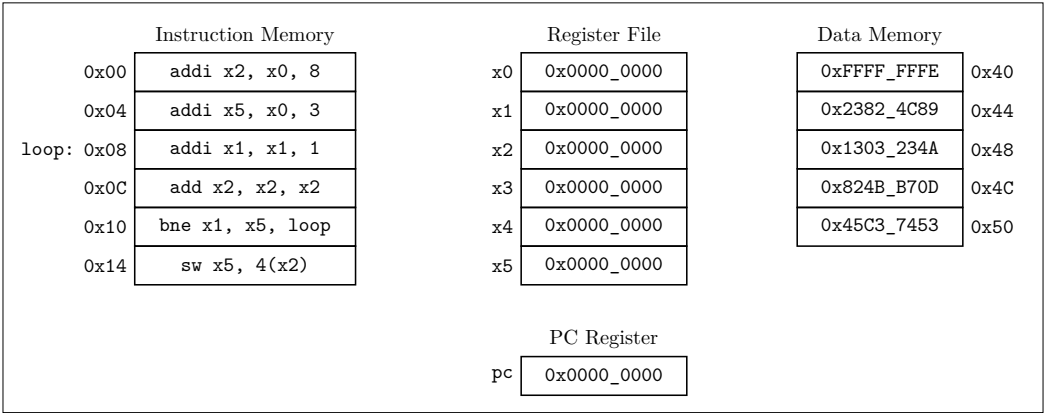
- 存储指令！指令可以用 32 位二进制编码，和数据一样存储在内存中。
- 存储指令的内存空间一般称为 IMEM（Instruction Memory）。
- 存储数据的内存空间一般称为 DMEM（Data Memory）。
- 处理器中存在一个特殊的 PC 寄存器（PC Register），记为 `pc`，它独立于寄存器堆中的 32 个寄存器 `x0`、`x1`、`x2`、⋯、`x31`，专门用于存储指令的地址。
- 处理器在每个周期中，会根据 `pc` 从 IMEM 中读取出一条指令。
- 在正常的情况下，`pc` 会被更新为 `pc + 4`，实现顺序执行。
- 在跳转的情况下，`pc` 会被写入跳转目标指令的地址，实现跳转指令。

分支指令示例

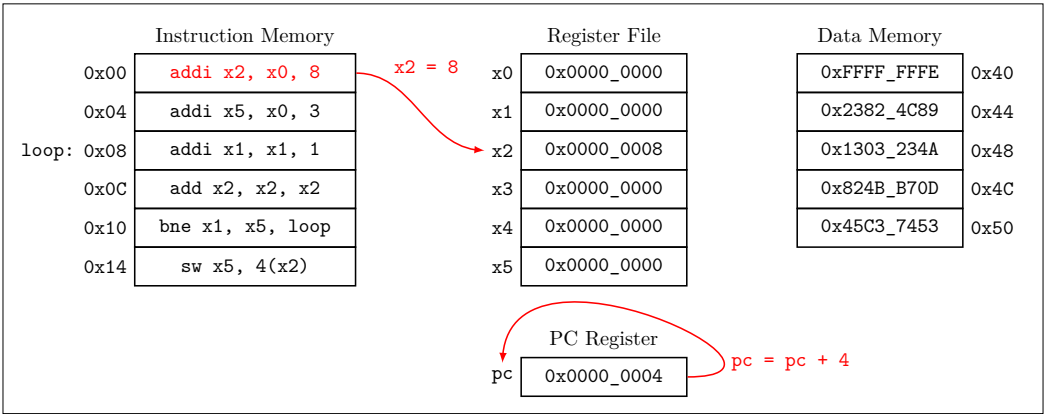
代码 14: 分支指令示例

```
1  addi x2, x0, 8              # x2 = 8
2  addi x5, x0, 3              # x5 = 0
3  loop:                       # loop
4  addi x1, x1, 1              # x1 = x1 + 1
5  add x2, x2, x2              # x2 = x2 + x2
6  bne x1, x5, loop            # if x1 != x5: -> loop
7  sw x5, 4(x2)                # mem[x2 + 4] = x5
```

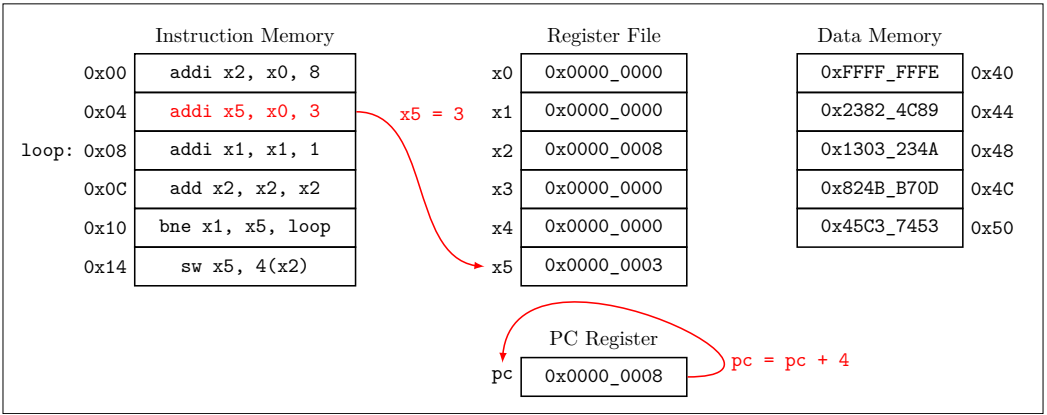
分支指令示例：初始状态


Instruction Memory
0x00 addi x2, x0, 8
0x04 addi x5, x0, 3
loop: 0x08 addi x1, x1, 1
0x0C add x2, x2, x2
0x10 bne x1, x5, loop
0x14 sw x5, 4(x2)
Register File
x0 0x0000_0000
x1 0x0000_0000
x2 0x0000_0000
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0000
PC Register
pc 0x0000_0000
Data Memory
0xFFFF_FFFE 0x40
0x2382_4C89 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50

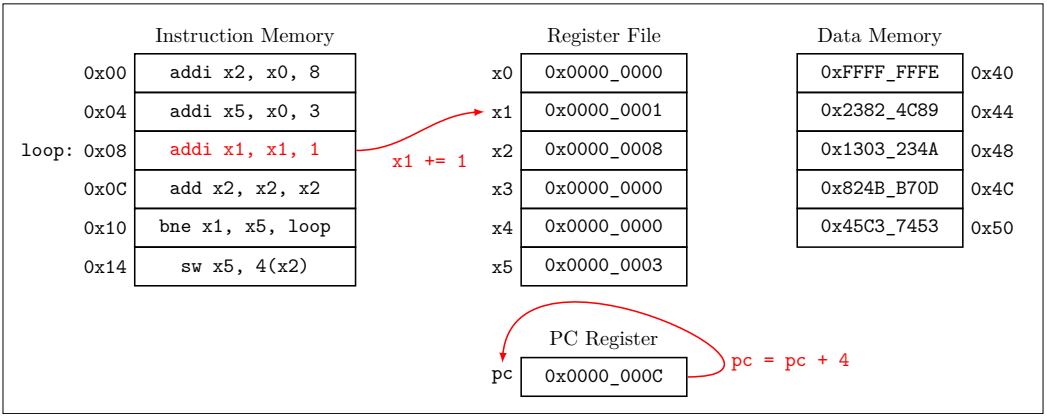
分支指令示例：周期 01


Instruction Memory
0x00 addi x2, x0, 8
0x04 addi x5, x0, 3
loop: 0x08 addi x1, x1, 1
0x0C add x2, x2, x2
0x10 bne x1, x5, loop
0x14 sw x5, 4(x2)
x2 = 8
Register File
x0 0x0000_0000
x1 0x0000_0000
x2 0x0000_0008
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0000
Data Memory
0xFFFF_FFFE 0x40
0x2382_4C89 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50
PC Register
pc 0x0000_0004
pc = pc + 4

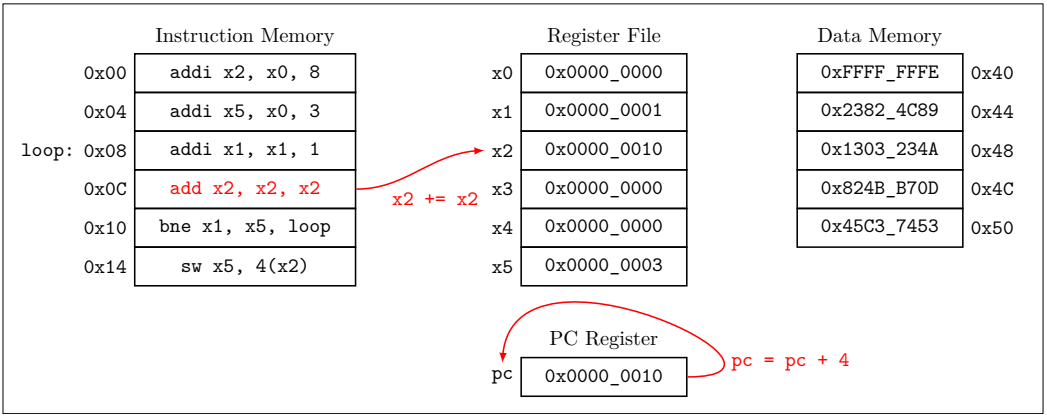
分支指令示例：周期 02


Instruction Memory
0x00 addi x2, x0, 8
0x04 addi x5, x0, 3
loop: 0x08 addi x1, x1, 1
0x0C add x2, x2, x2
0x10 bne x1, x5, loop
0x14 sw x5, 4(x2)
x5 = 3
Register File
x0 0x0000_0000
x1 0x0000_0000
x2 0x0000_0008
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0003
Data Memory
0xFFFF_FFFE 0x40
0x2382_4C89 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50
PC Register
pc 0x0000_0008
pc = pc + 4

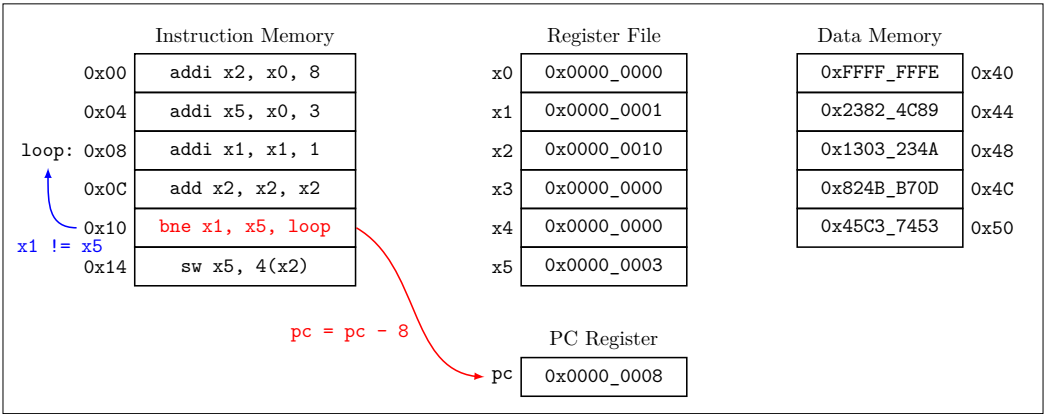
分支指令示例：周期 03（第一次循环）



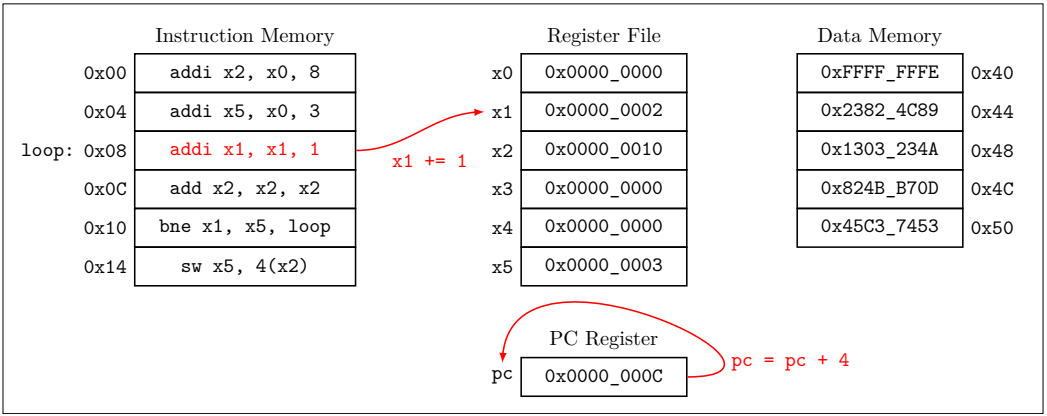
分支指令示例：周期 04（第一次循环）



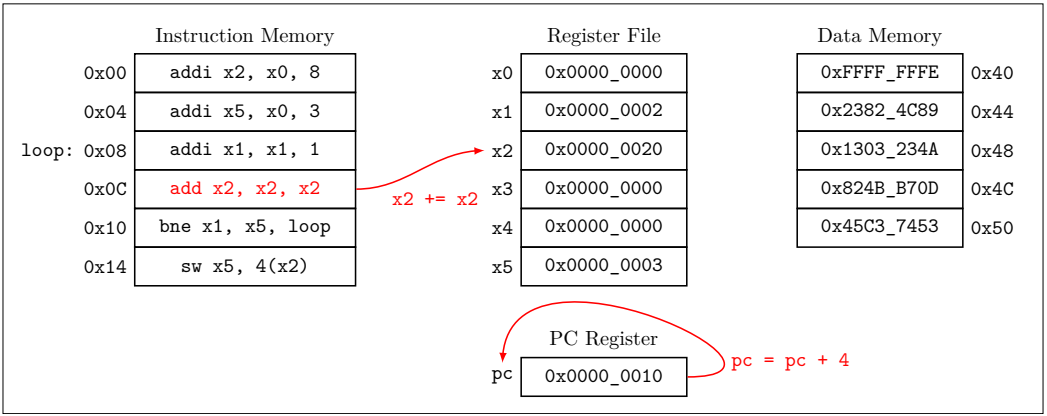
分支指令示例：周期 05（第一次循环）



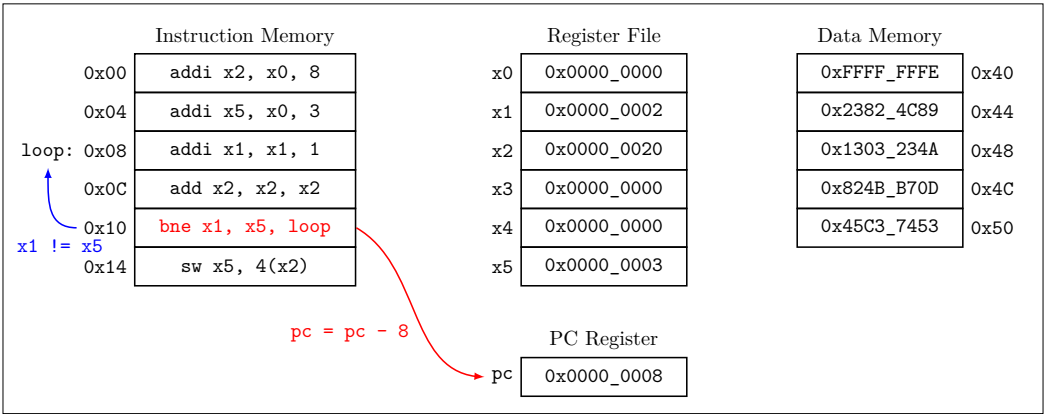
分支指令示例：周期 06（第二次循环）



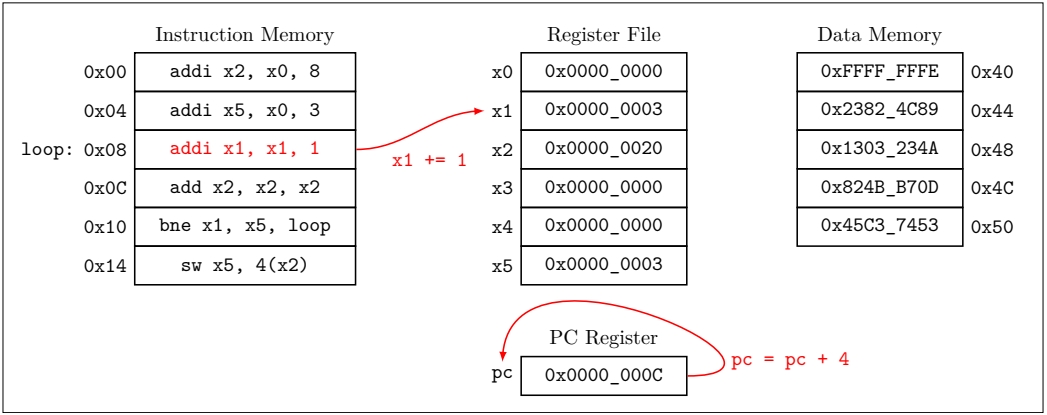
分支指令示例：周期 07（第二次循环）



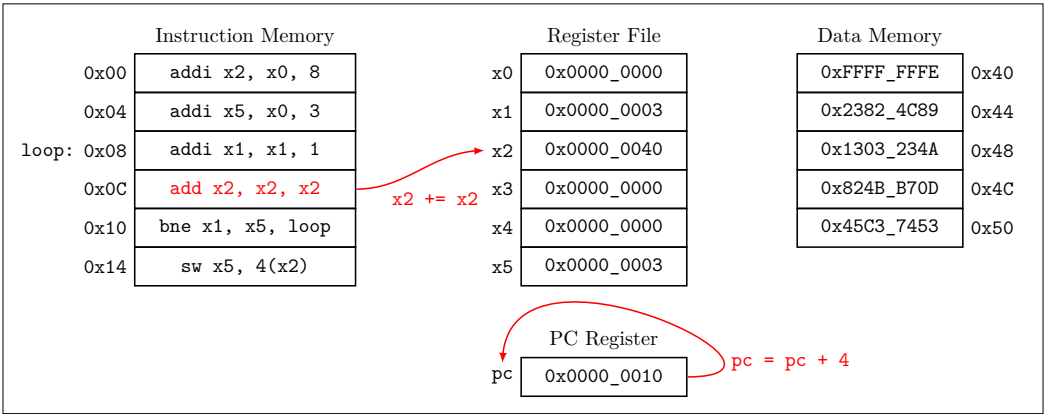
分支指令示例：周期 08（第二次循环）



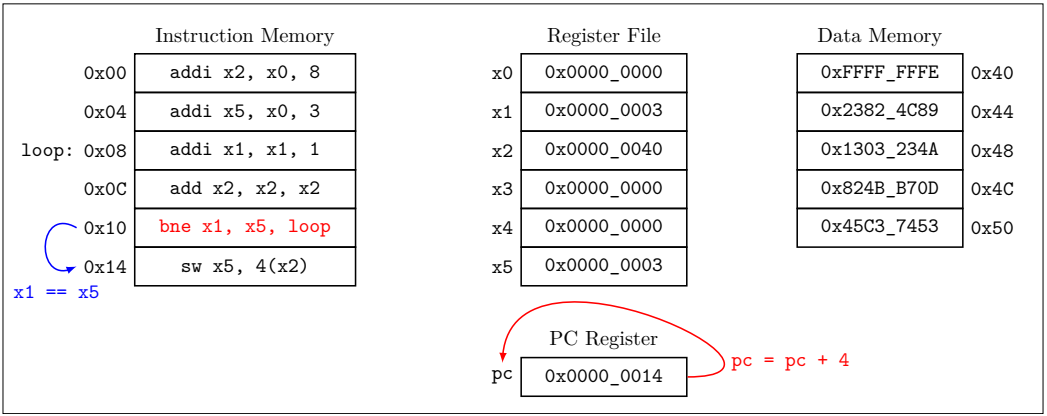
分支指令示例：周期 09（第三次循环）



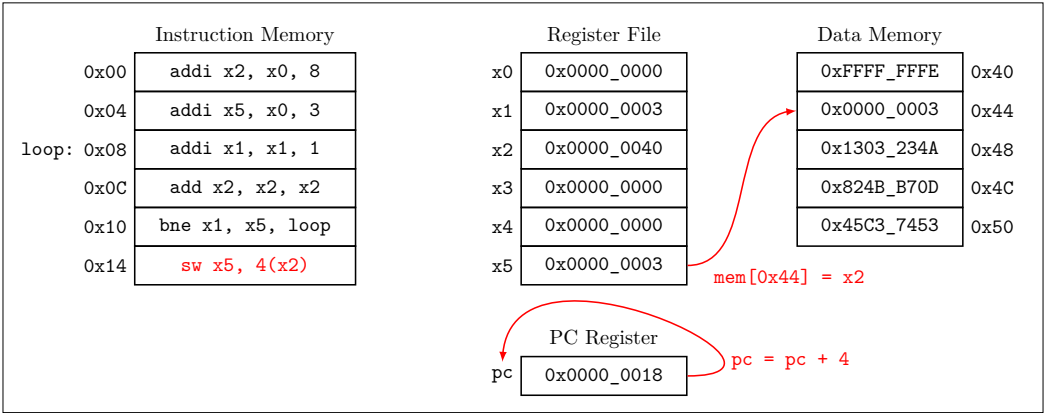
分支指令示例：周期 10（第三次循环）



分支指令示例：周期 11（第三次循环）



分支指令示例：周期 12


Instruction Memory
0x00 addi x2, x0, 8
0x04 addi x5, x0, 3
loop: 0x08 addi x1, x1, 1
0x0C add x2, x2, x2
0x10 bne x1, x5, loop
0x14 sw x5, 4(x2)
Register File
x0 0x0000_0000
x1 0x0000_0003
x2 0x0000_0040
x3 0x0000_0000
x4 0x0000_0000
x5 0x0000_0003
Data Memory
0xFFFF_FFFE 0x40
0x0000_0003 0x44
0x1303_234A 0x48
0x824B_B70D 0x4C
0x45C3_7453 0x50
mem[0x44] = x2
PC Register
pc 0x0000_0018
pc = pc + 4

分支指令的总结

分支指令的本质也是在做计算，只不过计算以修改 PC 寄存器的方式实现控制流

- 分支指令中，对于 `bne`，它实际上要做减法，并根据结果是否非零进行跳转。
- 分支指令中使用标签（例如 `loop`）作为跳转目标，然而，这些标签并不会出现在指令的编码中。汇编器需要完成的一个任务就是将标签转换为指令的地址偏移。例如 `0x10` 处的 `bne x1, x5, loop` 是要跳转到 `0x08` 处，这个地址偏移就应当是 `-8`，最终反映在 `pc = pc - 8` 上。这里的 `-8` 实际上也是一种立即数。

无条件的跳转

有时，跳转的发生是无条件的

- 有条件的跳转：使用分支指令，例如 `bne` 指令。
- 无条件的跳转：使用跳转指令，例如 `jal` 指令。

跳转指令 jal

跳转指令（Jump and Link）

代码 15: 跳转指令

```
1 addi x1, x0, 8              # x1 = 8
2 addi x2, x0, 8              # x2 = 8
3 jal x5, target              # -> target, x5 = pc + 4
4 addi x3, x0, 73             # x3 = 73 (skip)
5 target:
6 addi x4, x0, 74             # x4 = 74
```

跳转指令 jal

RISC-V 中，为什么跳转指令要称为“跳转并链接”？

- 跳转指令经常是用于调用某个函数，简单来说，就是要从主程序处跳转至别处执行若干指令后再跳转回主程序。因此，跳转前，需将 pc + 4 存储到一个寄存器中，以便稍后能再跳转返回这里，这个行为就称为链接（Link）。
- 跳转指令的链接，本质上就是在为跳转设置一个“返回点”。
- 跳转指令如果不需要链接，可以简单的用 jal x0，target 将 pc + 4 丢弃。
- 除了 jal，还有一条 jalr 指令用于跳转到某个寄存器记录的地址处。

PC

PC 寄存器如图3所示

- 只是一个普通的 32 位寄存器。
- 右侧读出当前周期要执行的指令地址。
- 左侧写入下一周期要执行的指令地址。
- 对于顺序执行，写入 `pc + 4`。
- 对于跳转执行，写入跳转的目标指令地址。

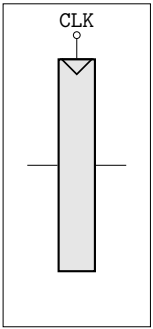


图 3: PC

IMEM

IMEM 的端口设计如图4所示

- A 是指令地址，传入 pc 即可。
- RD 是读出的指令。
- 只读！处理器不需要向 IMEM 写入任何内容。

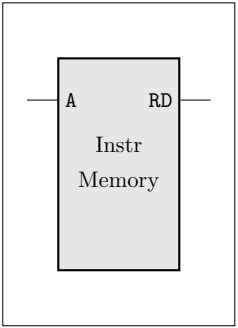


图 4: Instruction Memory

5 总结

总结

至此，通过上述对 RISC-V 指令的分析，处理器所需的四个和时序逻辑相关的存储模块：**PC**、**REGFILE**、**IMEM**、**DMEM**，都已经齐备了！距离一个完整可运作的处理器，只需要再往这些存储模块之间添加一点点“简单”的组合逻辑就可以了。

谢谢大家！