SI100B EE-Part MiniCPU

RISC-V Microarchitecture

Yuxuan Li

VSPLab@SIST

2026.06.05

liyx22025@shanghaitech.edu.cn

目录

1 三种微架构的设计思路

2 处理器的关键：1 个 ALU，2 个 ADD，3 个 MUX

3 处理器的数据通路

4 控制单元设计

5 总结

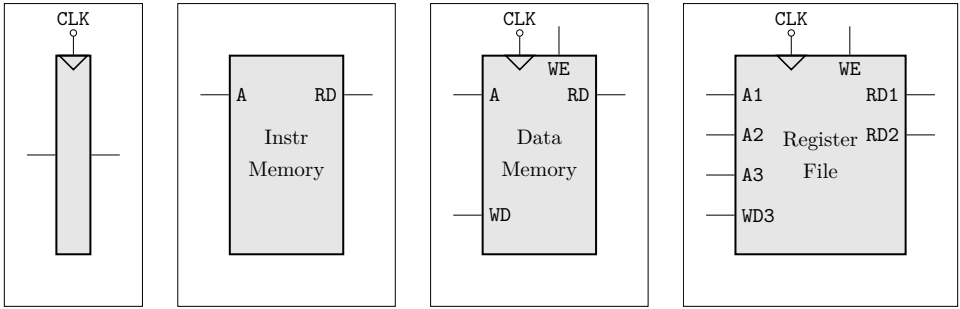
1 三种微架构的设计思路

三种微架构的设计思路

本课程只探讨单周期处理器的设计

- 单周期处理器：指令在一个周期内完成执行。
- 多周期处理器：指令在多个周期内完成执行。
- 流水线处理器：引入流水线架构。

四个状态单元

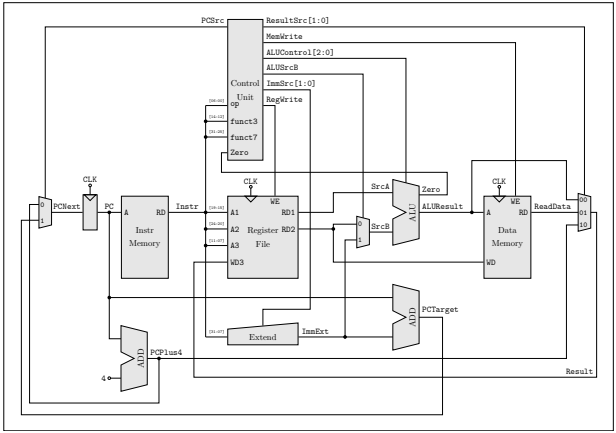


(a) PC (b) IMEM (c) DMEM (d) REGFILE

图 1: RISC-V 处理器的四个状态单元

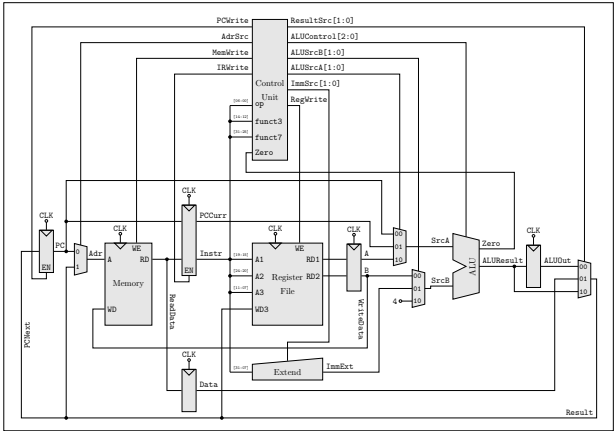
单周期处理器

- 执行一条指令需要一个周期
- 执行最慢的指令决定最小周期
- 最简单的微架构设计



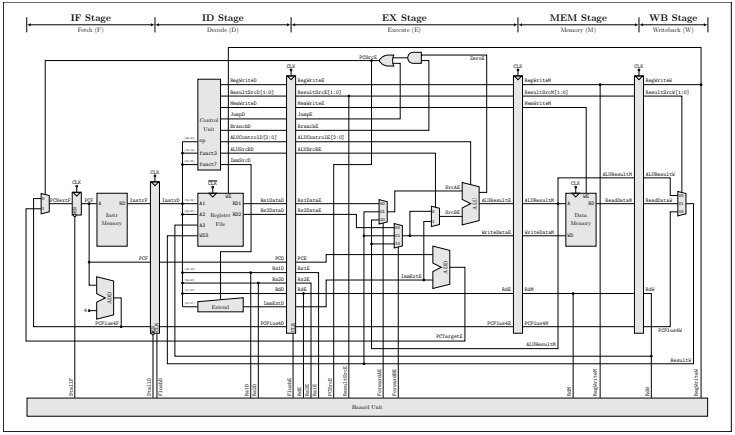
多周期处理器

- 执行一条指令需要多个周期
- 简单的指令需要较少周期
- 复杂的指令需要较多周期
- 模块被更多的复用
- 需要维护非架构状态



流水线处理器

- 流水线：将单周期拆分为五个阶段，各阶段间插入寄存器，以同时执行五条指令。
- 取址（Fetch）
- 译码（Decode）
- 执行（Execute）
- 访存（Memory）
- 写回（Writeback）
- 需要处理流水线冲突



2 处理器的关键：1 个 ALU，2 个 ADD，3 个 MUX

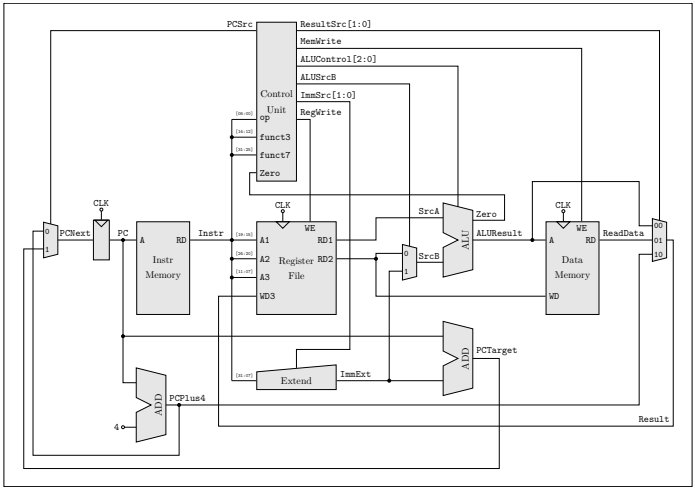


图 2: RISC-V 单周期处理器

处理器的关键：1 个 ALU，2 个 ADD，3 个 MUX

尽管图2所示的处理器电路相当复杂，但理解其原理，只需要把握住

- 1 个 ALU
- 2 个 ADD：即 PCPlus4ADD、PCTargetADD
- 3 个 MUX：即 PCNextMUX、SrcBMUX、ResultMUX

1 个 ALU

ALU 的作用是完成处理器中最核心的计算，在不同指令中

- 寄存器算术指令：计算 `rs1 + rs2` 或 `rs1 & rs2` 这样的各类算术运算。
- 立即数算术指令：计算 `rs1 + imm` 或 `rs1 & imm` 这样的各类算术运算。
- 读写指令：计算 `imm(rs1)` 即 `rs1 + imm` 的 DMEM 地址偏移。
- 分支指令：计算 `rs1 - rs2` 以判断是否要进行跳转。
- 跳转指令：无需进行任何计算！

2 个 ADD

ALU 无法完成所有的计算，有两个专用的 ADD 辅助完成指令地址的计算

- PCPlus4ADD：完成顺序执行下指令地址 `pc + 4` 的计算。
- PCTargetADD：完成跳转执行下指令地址 `pc + imm` 的计算。

3 个 MUX

MUX 是多路选择开关，它是实现处理器能处理多样指令的关键

- PCNextMUX：决定写入 PC 寄存器的指令地址是 `pc + 4` 还是 `pc + imm`。
- SrcBMUX：决定参与 ALU 计算的操作数是 `rs1`，`rs2` 还是 `rs1`，`imm`。
- ResultMUX：决定写回 REGFILE 的数据来源：ALU、MEM，PCPlus4ADD?

3 处理器的数据通路

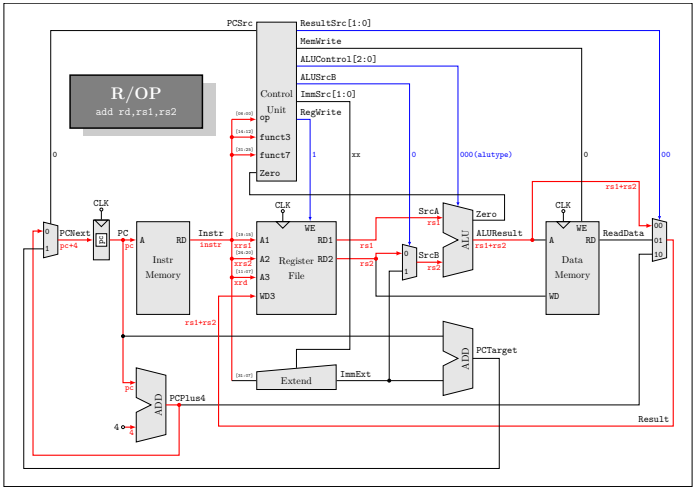


图 3: 寄存器算术指令的数据通路

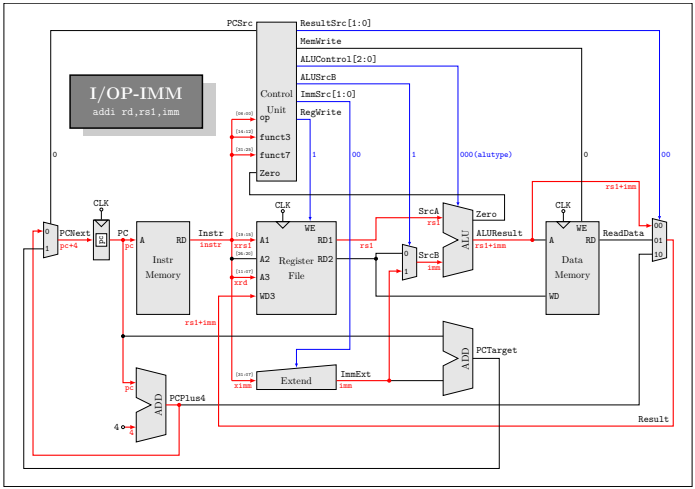


图 4: 立即数算术指令的数据通路

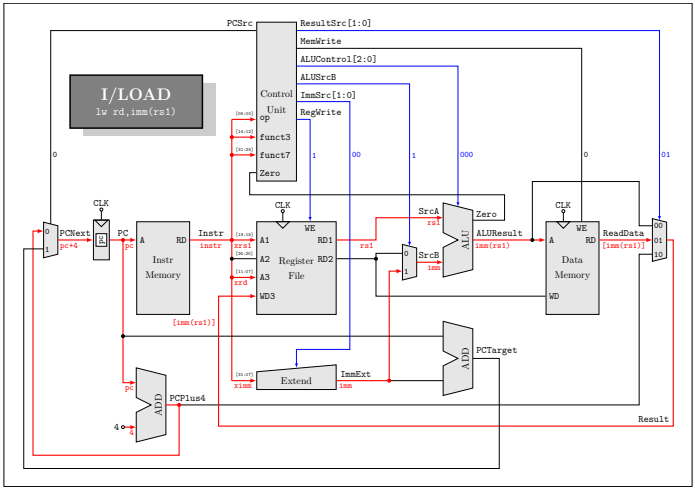

I/LOAD
lw rd,imm(rs1)
PCSrc
ResultSrc[1:0]
MemWrite
ALUControl[2:0]
ALUSrcB
ImmSrc[1:0]
RegWrite
Control Unit
op
funct3
funct7
Zero
CLK
PCNext
pc+4
PC
Instr Memory
Instr
Register File
WE
A1
A2
A3
WD3
RD1
RD2
SrcA
SrcB
ALU
ALUResult
Zero
Data Memory
ReadData
WD
Extend
ImmExt
ADD
PCTarget
PCPlus4
Result

图 5: 读指令的数据通路

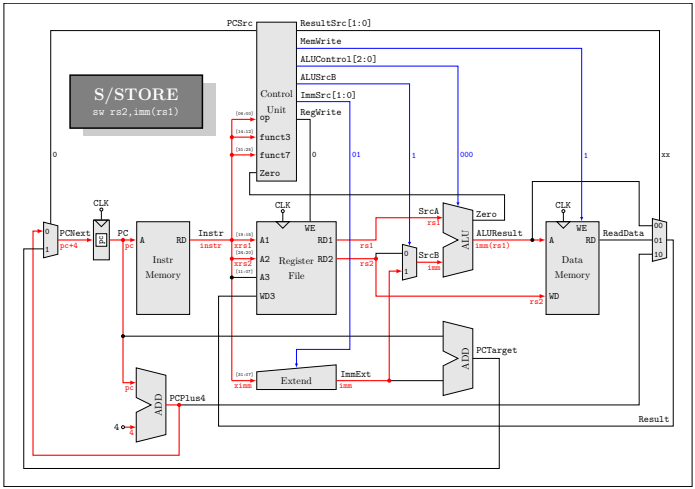

S/STORE
sw rs2,imm(rs1)
PCSrc
ResultSrc[1:0]
MemWrite
ALUControl[2:0]
ALUSrcB
ImmSrc[1:0]
RegWrite
Control Unit
op
funct3
funct7
Zero
CLK
PCNext
pc+4
PC
Instr Memory
Instr
Register File
WE
A1
A2
A3
WD3
RD1
RD2
SrcA
SrcB
ALU
ALUResult
Zero
Data Memory
ReadData
WD
PCTarget
ADD
Extend
ImmExt
PCPlus4
Result

图 6: 写指令的数据通路

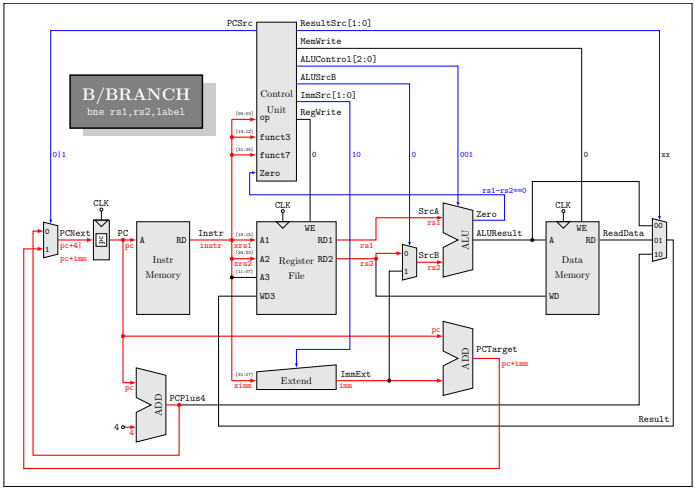


图 7: 分支指令的数据通路

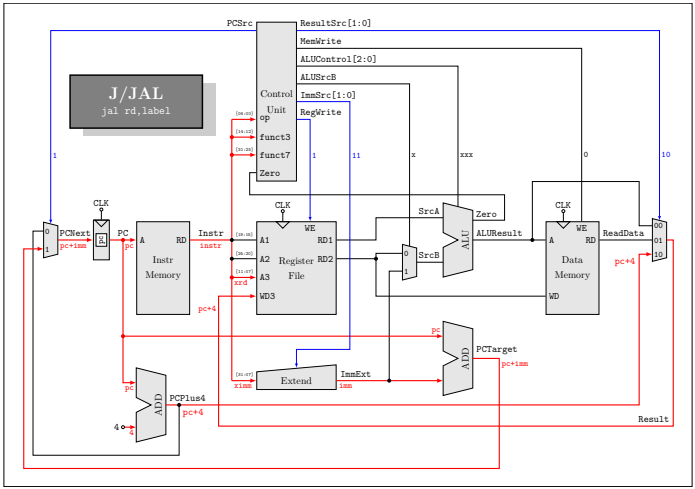
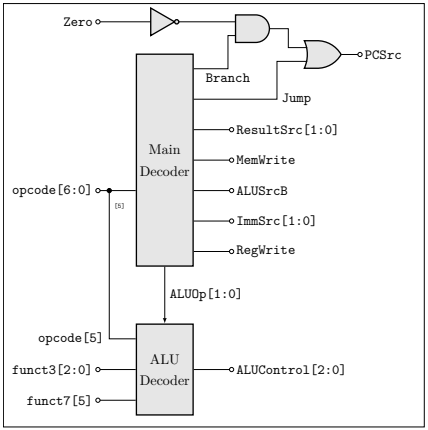


图 8: 跳转指令的数据通路

4 控制单元设计

控制单元

- 控制单元产生所有控制信号，分为两级。
- 第一级是 Main Decoder，仅根据 opcode 产生控制信号。第二级是 ALU Decoder，进一步根据 `funct3` 和 `funct7` 确定 ALU 的控制信号。
- 应指出，这种译码结构高度依赖本课程要求实现的子集的简单性，例如，读写指令只有 `lw` 和 `sw`，分支指令只有 `bne`，因此，几乎只有算术指令需要考虑 `funct3` 和 `funct7`。



控制信号 MemWrite

`MemWrite` 控制是否要写 DMEM

- S/STORE 是唯一可以对 DMEM 写入数据的指令类型，只有其使用该信号。
- 大部分指令都是在操作寄存器！
- 写地址一定是 `rs1 + imm`
- 写数据一定是 `rs2`

表 1: MemWrite 的产生

类型	opcode	MemWrite
I/LOAD	0000011	0
S/STORE	0100011	1
R/OP	0110011	0
I/OP-IMM	0010011	0
B/BRANCH	1100011	0
J/JAL	1101111	0

控制信号 RegWrite

`RegWrite` 控制是否要写 REGFILE

- B/BRANCH 和 S/STORE 是唯二不需要写回 REGFILE 的指令类型，前者进行分支决策，后者数据写到 DMEM。
- I/OP-IMM 和 R/OP 写 ALU 运算结果。
- I/LOAD 写 DMEM 读取结果。
- J/JAL 写 PCPlus4Add 的运算结果。

表 2: RegWrite 的产生

类型	opcode	RegWrite
I/LOAD	0000011	1
S/STORE	0100011	0
R/OP	0110011	1
I/OP-IMM	0010011	1
B/BRANCH	1100011	0
J/JAL	1101111	1

控制信号 ResultSrc

ResultSrc 控制写回 REGFILE 的数据来源

- 来自 ALU、DMEM、PCPlus4ADD?
- ResultSrc = 00：写回 ALUResult
- ResultSrc = 01：写回 ReadData
- ResultSrc = 10：写回 PCPlus4

表 3: ResultSrc 的产生

类型	opcode	ResultSrc
I/LOAD	0000011	01
S/STORE	0100011	xx
R/OP	0110011	00
I/OP-IMM	0010011	00
B/BRANCH	1100011	xx
J/JAL	1101111	10

控制信号 PCSrc

PCSrc 控制写入 PC 寄存器的指令地址

- 来自 PCPlus4ADD、PCTargetADD?
- PCSrc = 0：PCPlus4 = pc + 4
- PCSrc = 1：PCTarget = pc + imm
- 换言之，顺序执行还是跳转执行?
- PCSrc 依赖中间信号 Jump 和 Branch。
- 若 Jump 为高，或者 Branch 为高且来自 ALU 的 Zero 非零（只考虑 bne 的情况），则应当令 PCSrc = 1 跳转。

表 4: PcSrc 的产生

类型	opcode	Jump	Branch
I/LOAD	0000011	0	0
S/STORE	0100011	0	0
R/OP	0110011	0	0
I/OP-IMM	0010011	0	0
B/BRANCH	1100011	0	1
J/JAL	1101111	1	0

控制信号 ALUSrcB

ALUSrcB 控制 ALU 第二个操作数的来源

- ALUSrcB = 0：来自 rs2
- ALUSrcB = 1：来自 imm
- R/OP 对 rs1 和 rs2 进行计算。
- B/BRANCH 对 rs1 和 rs2 进行比较。
- 立即数是多数情况!

表 5: ALUSrcB 的产生

类型	opcode	ALUSrcB
I/LOAD	0000011	1
S/STORE	0100011	1
R/OP	0110011	0
I/OP-IMM	0010011	1
B/BRANCH	1100011	0
J/JAL	1101111	x

控制信号 ALUOp

ALUOp 是 ALUControl 的引导信号，其目标是确定 ALU 需要完成是算术指令的指定计算，还是某种特定的功能性的计算

- ALUOp = 00：加法（地址偏移）
- ALUOp = 01：减法（分支比较）
- ALUOp = 10：算术指令

表 6: ALUOp 的产生

类型	opcode	ALUOp
I/LOAD	0000011	00
S/STORE	0100011	00
R/OP	0110011	10
I/OP-IMM	0010011	10
B/BRANCH	1100011	01
J/JAL	1101111	xx

控制信号 ALUControl

表 7: ALUControl 的产生

指令	ALUOp	funct3	{op[5],funct7[5]}	ALUControl	说明
lw,sw	00	xxx	xx	000	加法
beq	01	xxx	xx	001	减法
add,addi	10	000	00|01|10	000	加法
sub	10	000	11	001	减法
and,andi	10	111	xx	010	按位与
or,ori	10	110	xx	011	按位或
slt,slti	10	010	xx	101	当小于时赋值

控制信号 ALUControl

ALUControl 会根据 ALUOp 判明的初步结果，结合 opcode、funct3、funct7 最终确定 ALU 执行的具体计算类型，这可能有以下五种情况

- ALUControl = 000，ALU 做加法。
- ALUControl = 001，ALU 做减法。
- ALUControl = 010，ALU 做按位与。
- ALUControl = 011，ALU 做按位或。
- ALUControl = 101，ALU 做当小于时赋值。

控制信号 ALUControl

特别要注意的是 `funct3 = 000` 的算术指令在判断加减法的问题

- 立即数算术指令 `opcode = 0010011`，差异点是 `opcode[5] = 0`
- 寄存器算术指令 `opcode = 0110011`，差异点是 `opcode[5] = 1`
- 若为立即数算术指令，其没有 `funct7`，故无论 `funct7[5]` 取值，都是 `addi`。
- 若为寄存器算术指令，`funct7 = 0000000` 即 `funct7[5] = 0` 为 `add`。
- 若为寄存器算术指令，`funct7 = 0100000` 即 `funct7[5] = 1` 为 `sub`。

控制信号 ImmSrc

`ImmSrc` 是立即数扩展单元的控制信号

- `ImmSrc = 00`：立即数类型为 I 型
- `ImmSrc = 01`：立即数类型为 S 型
- `ImmSrc = 10`：立即数类型为 B 型
- `ImmSrc = 11`：立即数类型为 J 型
- R 型指令没有立即数，不需要扩展！

表 8: ImmSrc 的产生

类型	opcode	ImmSrc
I/LOAD	0000011	00
S/STORE	0100011	01
R/OP	0110011	xx
I/OP-IMM	0010011	00
B/BRANCH	1100011	10
J/JAL	1101111	11

控制信号 ImmSrc

表 9: ImmSrc 的效果

类型	ImmSrc	ImmExt	说明
I	00	{20{I[31]},I[31:20]}	12 bit
S	01	{20{I[31]},I[31:25],I[11:7]}	12 bit
B	10	{20{I[31]},I[07],I[30:25],I[11:08],1'b0}	13 bit
J	11	{12{I[31]},I[19:12],I[20],I[30:21],1'b0}	21 bit

5 总结

总结

至此，所有要素都齐备了，开始搭建自己的处理器吧！

谢谢大家！