体系结构——微架构

2025 年 12 月 30 日

摘要

在本章中，我们将研究三种 arm 处理器微结构：单周期，多周期和流水线。他们的区别在于状态元件的连接方式和非体系结构状态的数量。

1 微架构考点

1. **处理器性能分析**
   掌握执行时间公式、CPI (Cycles Per Instruction)、以及时钟周期 T_c 的定义与计算方法。

2. **单周期处理器设计**
   理解关键路径分析与 T_c 的计算，以及 LDR/STR/分支指令的数据通路设计。

3. **多周期处理器**
   理解有限状态机 (FSM) 控制器的工作原理、指令的周期分解以及 CPI 的计算。

4. **流水线处理器基础**
   掌握五级流水线结构 (Fetch, Decode, Execute, Memory, Writeback) 及流水线寄存器的作用。

5. **数据冲突 (Data Hazards)**
   理解写后读 (RAW) 冲突的成因，以及如何通过数据转发 (Forwarding) 机制解决冲突。

6. **停顿**
   掌握 Load-Use 冒险 (Load-Use Hazard) 的检测逻辑以及如何插入停顿 (Stall) 来解决它。

7. **控制冲突 (Control Hazards)**
   理解分支指令导致的流水线冲突，以及如何通过清空 (Flush) 或流水线冒泡来处理。

8. **ARM 架构特性实现**
   理解条件执行逻辑 (Conditional Logic) 的硬件实现，以及 PC/R15 的特殊更新机制。

2 单周期数据通路

2.1 LDR 指令的数据通路构建 (Step-by-Step)

单周期处理器的设计从 LDR 指令开始，分为以下 6 个步骤逐步构建数据通路。

2.1.1 STEP 1: 取指 (Fetch Instruction)

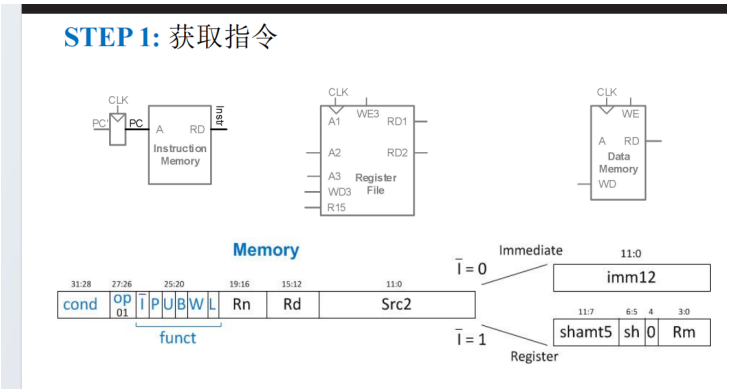

图 1: 取指阶段：PC 读取指令存储器并自增

- **操作**：PC 输出地址给指令存储器，取出指令 `Instr`。同时 PC 通过加法器计算 $PC + 4$。
- **目的**：获取当前指令并指向下一条指令。

2.1.2 STEP 2: 译码与读寄存器 (Read Registers)

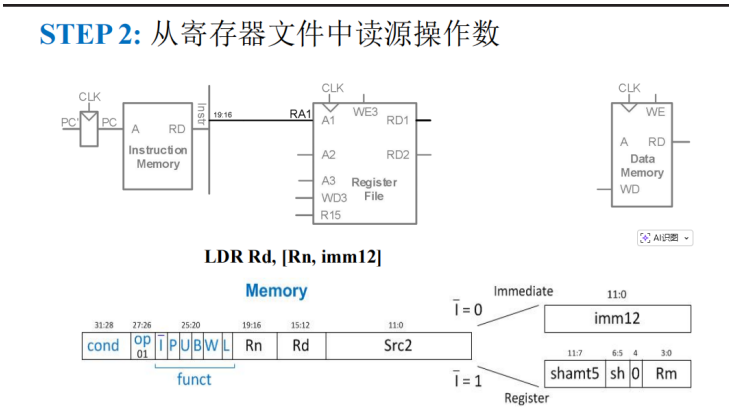

图 2: 译码阶段：根据 Rn 读取基址寄存器

- **操作**：将指令的 `19:16` 位 (Rn) 连接到寄存器堆的读端口 $A1$。
- **输出**：端口 $RD1$ 输出基址寄存器的值（例如 R2 的值）。

2.1.3 STEP 3: 扩展立即数 (Extend Immediate)

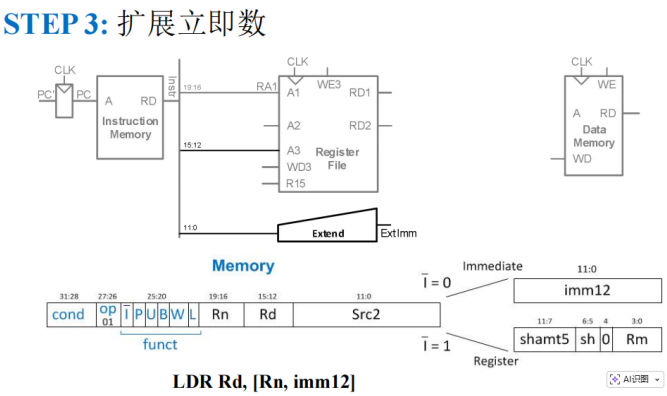

图 3: 扩展阶段：将 12 位立即数扩展为 32 位

- **操作**：指令中的低 12 位 ($Src2$) 输入扩展单元 (Extend)。
- **控制**：LDR 指令通常使用零扩展，输出 32 位的 $ExtImm$。

2.1.4 STEP 4: 计算地址 (Calculate Address)

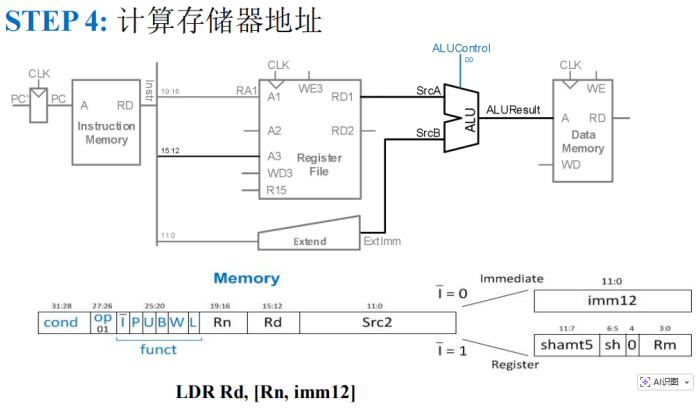

图 4: 执行阶段：ALU 计算内存目标地址

- **操作**：ALU 接收 $SrcA$ (基址) 和 $SrcB$ (扩展后的偏移量)。
- **控制**：$ALUControl$ 设为 00 (ADD)，计算 $Rn + Imm$。

2.1.5 STEP 5: 访存与写回 (Memory Writeback)

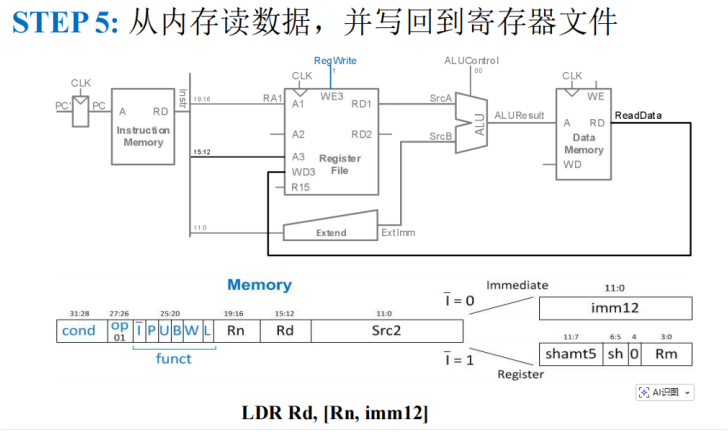

图 5: 访存阶段：读取数据并写入目标寄存器

- **访存**：$ALUResult$ 作为地址输入数据存储器 (Data Memory)，读出 $ReadData$。
- **写回**：$ReadData$ 连回寄存器堆的 $WD3$，地址为指令的 `15:12` (Rd)。需设置 $RegWrite = 1$。

2.1.6 STEP 6: PC 更新逻辑 (PC Update)

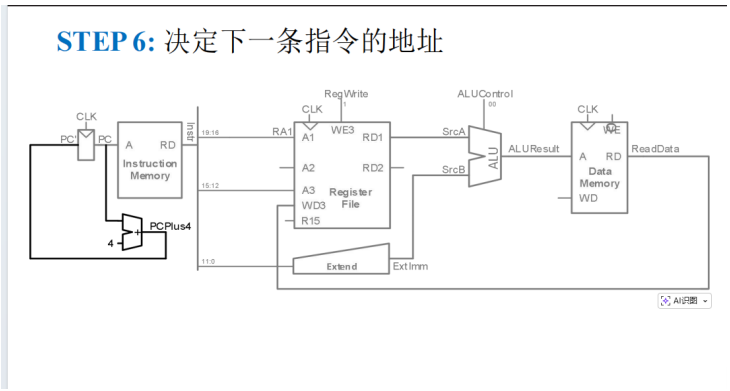

图 6: 更新 PC：准备下一条指令地址

- **操作**：将 STEP 1 计算好的 $PC + 4$ 写回 PC 寄存器。
- **扩展**：如果是分支指令，此处会引入多路选择器 (Mux) 来选择是 $PC + 4$ 还是跳转目标地址。

2.2 数据通路的扩展：数据处理与分支

在完成基础的 LDR/STR 数据通路构建后，我们需要对电路进行修改，以支持算术逻辑运算和程序跳转。这主要通过引入多路选择器（Mux）来改变数据的来源和去向来

实现。

2.2.1 数据处理指令 (Data-processing Instructions)

数据处理指令主要包括算术运算（如 ADD, SUB）和逻辑运算（如 AND, ORR）。这类指令的核心特点是：**计算结果来自 ALU，而不是存储器**。

根据第二个源操作数的不同，分为两种情况：

A. 立即数寻址 (Immediate Addressing) 例如指令：`ADD R1, R2, #5`

源操作数 A (SrcA)：

- 来自寄存器文件。指令中的 Rn 字段（位 19:16）指定地址，从端口 $RD1$ 读出数据（如 R2 的值）。

源操作数 B (SrcB) 的选择：

指令中的 8 位立即数（位 7:0）经过扩展单元（Extend Unit）处理。**关键 Mux**：ALU 输入端的 **ALUSrc Mux** 选择通道 1。扩展单元根据 `ImmSrc=00` 进行 8 位零扩展，生成 32 位操作数。

执行 (Execute)：

ALU 执行加法运算：$Result = SrcA + ExtImm$。

写回 (Writeback)：

关键 Mux：数据存储器之后的 **MemtoReg Mux** 选择通道 0。这意味着写入寄存器的数据直接来自 `ALUResult`，而非内存读取的数据。结果被写回寄存器文件中的 Rd（位 15:12）。

B. 寄存器寻址 (Register Addressing) 例如指令：`ADD R1, R2, R3`

源操作数 A (SrcA):同样来自寄存器文件的 $RD1$ 端口（读取 Rn）。

源操作数 B (SrcB) 的选择：

第二个操作数来自寄存器文件。指令中的 Rm 字段（位 3:0）被送入寄存器文件的读端口 $A2$。寄存器文件从 $RD2$ 端口输出 R3 的值。**关键 Mux**：**ALUSrc Mux** 选择通道 0，允许寄存器值进入 ALU，而不是立即数。

执行与写回：

ALU 计算 $SrcA + SrcB$。结果通过 `MemtoReg Mux` (通道 0) 写回目标寄存器 Rd 。

数据处理指令总结

ALUSrc：控制 ALU 第二个操作数是寄存器 (0) 还是立即数 (1)。**MemtoReg**：控制写回数据是 ALU 结果 (0) 还是内存数据 (1)。**RegWrite**：必须置为 1，以允许结果保存。

2.2.2 3.2 分支指令 (Branch Instructions)

分支指令（如 `B target`）用于改变程序执行流。它不操作通用寄存器或内存，而是修改 **PC (程序计数器)** 的值。

1. 目标地址计算 (Target Calculation) ARM 架构中，分支跳转地址的计算公式为：

$$\text{BranchTarget} = (PC + 8) + \text{SignExtended}(Imm24 \ll 2) \tag{1}$$

] 在单周期硬件中，这个计算复用了主 ALU ：

1. **获取基准地址 (PC+8)**：
   - 在 ARM 中，读取 R15 (PC) 得到的值是当前指令地址加 8。
   - 数据通路将寄存器文件的读端口 $A1$ 硬连线到 R15 (PC)。
   - 为了实现这一点，引入了 **RegSrc Mux**：对于分支指令，它选择 15 作为读取地址，而不是指令中的 Rn 。
2. **获取偏移量 (Offset)**：
   - 指令中的 24 位立即数（位 23:0）进入扩展单元。
   - 控制信号 `ImmSrc=10` 指示扩展单元进行 “符号扩展并左移 2 位” 的操作，生成 32 位偏移量。
   - **ALUSrc Mux** 选择 1，将该偏移量送入 ALU。
3. **ALU 加法**：
   - ALU 执行 $SrcA(PC + 8) + SrcB(Offset)$，计算出跳转目标地址。

2. 更新 PC (Update PC) 计算出的目标地址需要写入 PC 寄存器：

- **PCSrc Mux**：位于 PC 寄存器的输入端。
  - **通道 0**：来自 $PC + 4$ 加法器（用于顺序执行）。
  - **通道 1**：来自 ALU 的计算结果（用于分支跳转）。
- **控制逻辑**：

- 如果是无条件跳转（B），或者条件跳转满足（如 BEQ 且 $Z = 1$），控制单元设置 PCSrc = 1。
- PC 在下一个时钟上升沿更新为跳转地址。

分支指令总结

- **RegSrc**：选择读取 R15 (PC+8)。
- **ALUSrc**：选择立即数偏移量。
- **PCSrc**：选择 ALU 的结果作为新的 PC 值。
- **RegWrite / MemWrite**：均为 0，不修改数据状态。

2.3 控制单元详解 (Control Unit Details)

控制单元负责根据指令（Instruction）和状态标志（Flags）生成控制信号。它分为两级结构：译码器 (Decoder) 和条件逻辑 (Conditional Logic)。

2.3.1 主译码器 (Main Decoder)

主译码器根据指令的 *Op* (位 27:26) 和 *Funct* (位 25:20) 生成主要的控制信号。

表 1: 主译码器真值表

指令类型	Op	Funct_5 (I)	Funct_0 (L)	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
DP Reg	00	0	X	0	0	0	0	XX	1	00	1
DP Imm	00	1	X	0	0	0	1	00	1	X0	1
STR	01	X	0	0	X	1	1	01	0	10	0
LDR	01	X	1	0	1	0	1	01	1	X0	0
B	10	X	X	1	0	0	1	10	0	X1	0

关键信号解析：

- **ALUOp**：
  - 1：表示是数据处理指令，具体运算由 ALU Decoder 决定。
  - 0：表示是非数据指令（LDR/STR/B），强制 ALU 执行加法（计算地址）。
- **RegSrc**：
  - 10（STR）：端口 RA2 读取 Rd（位 15:12）作为写入内存的数据。
  - $X1$（B）：端口 RA1 读取 15（R15/PC）作为基地址。

2.3.2 ALU 译码器 (ALU Decoder)

ALU 译码器根据 $ALUOp$ 和指令的 $Funct$ 字段生成 $ALUControl$ 和 $FlagW$。

表 2: ALU 译码器真值表

ALUOp	**Funct$_{4:1}$ (cmd)**	**Funct$_0$ (S)**	**指令**	**ALUControl**	**FlagW**	**功能**
0	X	X	Not DP	00 (Add)	00	地址计算 (LDR/STR/B)
1	0100	0	ADD	00 (Add)	00	加法，不更标志
1	0100	1	ADDS	00 (Add)	11	加法，更新 NZCV
1	0010	0	SUB	01 (Sub)	00	减法，不更标志
1	0010	1	SUBS	01 (Sub)	11	减法，更新 NZCV
1	0000	0	AND	10 (And)	00	与，不更标志
1	0000	1	ANDS	10 (And)	10	与，只更新 NZ
1	1100	0	ORR	11 (Or)	00	或，不更标志
1	1100	1	ORRS	11 (Or)	10	或，只更新 NZ

2.3.3 PC 逻辑 (PC Logic)

PC 逻辑负责检测指令是否显式修改了程序计数器。

$$PCS = ((Rd == 15)\ \&\ RegW) \mid Branch \tag{2}$$

- **Branch**：分支指令直接修改 PC。
- **Implicit Write**：如果目标寄存器 Rd 是 15 且 $RegW$ 有效（例如 `MOV PC, LR`），也视为修改 PC。

2.3.4 条件逻辑 (Conditional Logic)

条件逻辑位于译码器之后，充当“守门员”。它根据当前状态标志 $Flags$ 和指令条件码 $Cond$ 决定指令是否生效。

条件检查 (Condition Check)

- 检查 $Cond$（指令高 4 位）与 $Flags$（NZCV）是否匹配。
- 输出内部信号 **CondEx** (Condition Executed)：匹配为 1，否则为 0。

信号门控 (Signal Gating) 如果条件不满足 ($CondEx = 0$)，所有的写信号被强制关闭，指令变为 NOP (空操作) 。

$$PCSrc = PCS\ \&\ CondEx \tag{3}$$

$$RegWrite = RegW\ \&\ CondEx \tag{4}$$

$$MemWrite = MemW\ \&\ CondEx \tag{5}$$

标志位更新控制 (Flag Updates) 状态寄存器 CPSR 的更新需要同时满足两个条件：

1. 指令要求更新（S 位为 1，即 $FlagW$ 有效）。
2. 指令条件满足执行（$CondEx = 1$）。

$$FlagWrite = FlagW \ \& \ CondEx \tag{6}$$

其中 $FlagWrite[1]$ 控制 N/Z 位更新，$FlagWrite[0]$ 控制 C/V 位更新。

2.3.5 CMP 控制单元的详细设计 (Control Unit)

CMP 的核心实现在于控制单元，它引入了一个关键信号：**NoWrite**。

译码器阶段 (Decoder) 当处理器获取到 CMP 指令时，译码器内部发生如下过程：

1. **主译码器 (Main Decoder)**：
   - 读取 `Op = 00` (数据处理指令)。
   - 按照通用规则，它认为需要写回结果，因此输出 **`RegW = 1`**。
2. **ALU 译码器 (ALU Decoder)**：
   - 检查 `Funct` 字段，发现 `cmd = 1010` (CMP 的特征码) 且 `S = 1`。
   - **动作 1**：输出 `ALUControl = 01` (执行减法)。
   - **动作 2**：输出 `FlagW = 11` (允许更新 NZCV 所有标志)。
   - **动作 3 (关键)**：识别出这是比较指令，输出 **`NoWrite = 1`**。

表 3: CMP 指令控制信号真值表

指令	ALUOp	Cmd	S	ALUControl	FlagW	NoWrite
ADD	1	0100	0	00 (Add)	00	0
SUBS	1	0010	1	01 (Sub)	11	0
CMP	**1**	**1010**	**1**	**01 (Sub)**	**11**	**1**

条件逻辑阶段 (Conditional Logic) 这是控制信号的汇聚点。如图所示，条件逻辑单元接收来自译码器的信号，并结合当前状态生成最终控制信号。

A. 写使能门控 (Write Gating) 为了防止 CMP 修改寄存器，`RegWrite` 的生成逻辑增加了一个反相输入：

$$\text{RegWrite} = \text{RegW}\ \textbf{AND}\ \text{CondEx}\ \textbf{AND}\ (\textbf{NOT}\ \text{NoWrite}) \tag{7}$$

逻辑推导：

- 主译码器请求写入：$RegW = 1$
- 假设条件满足：$CondEx = 1$
- ALU 译码器禁止写入：$NoWrite = 1 \rightarrow \textbf{NOT}\ NoWrite = 0$
- **最终结果**：$1\ \textbf{AND}\ 1\ \textbf{AND}\ 0 = \mathbf{0}$

因此，寄存器堆的写使能端接收到低电平，ALU 计算出的结果被丢弃。

B. 标志位更新 (Flag Update) 虽然结果被丢弃，但状态必须更新。

$$\text{FlagWrite} = \text{FlagW}\ \textbf{AND}\ \text{CondEx} \tag{8}$$

- 对于 CMP，`FlagW = 11`，只要条件满足 ($CondEx = 1$)，标志位就会被更新。

总结 CMP 指令的实现展示了硬件设计中的**“例外处理”**思想：

1. **主译码器**负责处理“一般情况”（Op=00 通常意味着要写寄存器）。
2. **ALU 译码器**负责识别“特殊情况”（CMP 不需要写寄存器），并发出 `NoWrite` 信号。
3. **条件逻辑**作为最后一级门控，综合所有条件，物理上切断了写使能信号。

2.4 单周期处理器性能分析 (Performance Analysis)

2.4.1 性能核心公式

衡量处理器性能的“铁三角”公式如下：

$$\text{Execution Time} = \text{Instruction Count} \times \text{CPI} \times T_{cycle} \tag{9}$$

在**单周期处理器**中，这三个变量具有以下特征：

- **Instruction Count (指令数)**：由程序本身决定。
- **CPI (Cycles Per Instruction)**：**固定为 1**。因为每条指令都在一个时钟周期内完成。CPI 会在多周期性能分析的时候有不同的取值。
- T_{cycle} **(时钟周期)**：这是性能的瓶颈所在。时钟周期必须足够长，以容纳**最慢**的那条指令（即关键路径）的执行时间。

表 4: 硬件元件延迟参数表

元件	参数	延迟 (ps)
寄存器输出延迟	t_{pcq_PC}	40
多路选择器	t_{mux}	25
ALU	t_{ALU}	120
译码器	t_{dec}	70
存储器读 (指令/数据)	t_{mem}	200
寄存器堆读	t_{RFread}	100
寄存器堆建立时间	$t_{RFsetup}$	60

2.4.2 关键路径分析 (Critical Path)

在 ARM 指令集中，**LDR (Load Register)** 指令通常决定了关键路径的长度，因为它经过的功能单元最多，路径最长。

LDR 指令的数据流向为：

1. **取指**：PC → 指令存储器 (t_{mem})
2. **译码/读寄存器**：译码器 + 寄存器堆读 ($t_{dec} + t_{RFread}$)
3. **执行**：ALU 计算地址 (t_{ALU})
4. **访存**：读取数据存储器 (t_{mem})
5. **写回**：将数据写入寄存器堆 ($t_{RFsetup}$)

关键路径延迟公式：

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup} \tag{10}$$

注：公式中包含 $2t_{mem}$（取指 + 访存）和 $2t_{mux}$（数据通路中的多路选择器延迟）。

2.4.3 性能计算示例 (Quantitative Example)

假设各元件的延迟参数如下表所示：

计算时钟周期 (T_c) 将上述参数代入关键路径公式：

$$\begin{aligned} T_c &= 40 + 2(200) + 70 + 100 + 120 + 2(25) + 60 \\ &= 40 + 400 + 70 + 100 + 120 + 50 + 60 \\ &= \mathbf{840}\ \text{ps} \end{aligned}$$

这意味着处理器的时钟频率上限约为 $1/840ps \approx 1.19$ GHz。

计算总执行时间 假设一个程序包含 **1000 亿** (100×10^9) 条指令，计算其运行时间：

$$\begin{aligned}\text{Execution Time} &= (\#\text{instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9) \times 1 \times (840 \times 10^{-12}\ \text{s}) \\ &= 84000 \times 10^{-3}\ \text{s} \\ &= \mathbf{84}\ \text{seconds}\end{aligned}$$

3 多周期数据通路 (Multi-Cycle)

3.1 PC 的访问与更新逻辑

在多周期处理器中，程序计数器 (PC/R15) 既是取指的地址源，也是通用的数据寄存器。设计必须解决 PC 的自增、作为操作数读取以及作为目标寄存器写入的问题。

3.1.1 PC 的自增 (Fetch 阶段)

在取指阶段 (Step 1)，不仅需要读取指令，还需要计算下一次的 PC 值。

- **目标**：更新 $PC \leftarrow PC + 4$。
- **硬件操作**：利用 ALU 进行加法运算。
- **控制信号**：
  - `ALUSrcA = 1`：选择 PC 作为源操作数 A。
  - `ALUSrcB = 01`：选择常数 4 作为源操作数 B。
  - `ALUOp = 00`：执行加法。
  - `PCWrite = 1`：使能 PC 写入，将 ALU 计算结果写回 PC。

3.1.2 读取 R15 (Read PC)

根据 ARM 架构规范，当 R15 作为源操作数参与运算时（例如 `ADD R1, R15, R2`），其值必须是当前指令地址 + 8。

- **问题**：在 Fetch 阶段结束时，PC 已经更新为 $PC + 4$。
- **解决方案**：
  1. 在 Step 1 (Fetch)，PC 更新为 $PC + 4$。
  2. 在 Step 2 (Decode)，ALU 再次被利用计算 $(PC + 4) + 4 = PC + 8$。
  3. 结果存入 `ALUResult`（或通过转发路径），供后续执行阶段使用。

3.1.3 写入 R15 (Write PC)

当目标寄存器是 R15 时（例如 `SUB R15, R8, R3`），运算结果直接修改程序计数器，实现跳转效果。

- **数据流**：指令的运算结果通过总线直接路由到 PC 的输入端。
- **控制**：控制器断言 `PCWrite` 信号，允许 ALU 的计算结果覆盖当前的 PC 值。

3.1.4 存储器写入指令 (STR) 的扩展

单周期实现 STR 较为简单，但在多周期架构中，由于指令和数据共用存储器且时序分步，需要引入临时寄存器。

- **核心动作**：$Mem[Address] \leftarrow Rd$。
- **硬件扩展**：
  - **Data 寄存器**：在译码阶段读取源寄存器 Rd 的值，并将其暂存在临时寄存器 `Data` 中（图中的 `WriteData` 信号来源）。这是为了防止在后续周期寄存器堆端口被挪作他用。
  - **ALUOut**：在执行阶段计算出的内存地址被保存在 `ALUOut` 中。
- **Step 5 (写入阶段)**：
  - `AdrSrc = 1`（或 IorD）：多路选择器选择 `ALUOut` 作为存储器地址，而非 PC。
  - `MemWrite = 1`：将 `Data` 寄存器的数据写入存储器。

3.2 数据处理指令 (Data Processing)

这两页对比了“立即数寻址”和“寄存器寻址”在硬件实现上的微小但关键的差异。

3.2.1 立即数寻址 (DP Immediate，第 88 页)

- **指令示例**：`ADD R1, R2, #5`
- **结论**：数据通路**不需要额外修改**。
- **流程**：源操作数 B 直接来自扩展后的立即数 `ExtImm`，ALU 计算结果存入 `ALUOut`，最终写回寄存器堆。

3.2.2 寄存器寻址 (DP Register)

- **指令示例**: `ADD R1, R2, R3`
- **硬件挑战**: 需要同时读取两个寄存器 Rn (Source 1) 和 Rm (Source 2)。
- **关键改进**: 引入 `RegSrc` 多路选择器。
  - **作用**: 控制寄存器堆的第二个地址输入端口 `RA2`。
  - **逻辑**:
    - 对于 STR 指令，`RA2` 读取的是指令的 Rd 字段 (Bits 15:12)，用于读取要存储的数据。
    - 对于 DP Register 指令，`RA2` 需要读取指令的 Rm 字段 (Bits 3:0)，作为第二个源操作数。
  - 因此，`RegSrc` 根据指令类型在 Rd 和 Rm 字段之间进行切换。

3.2.3 分支指令 (Branch)

第 90 页展示了分支指令 `B` 的最终实现，核心在于目标地址的计算。

- **目标地址计算 (BTA)**:

$$BTA = (SignExt(Imm24) \ll 2) + (PC + 8) \tag{11}$$

注: 在多周期硬件的具体实现中，通常利用 *Step 1* 已更新的 PC 值 (即 *PC+4*)，因此硬件操作通常是 $PC + 4 + Offset$ 或者根据具体微架构调整 ALU 输入。

- **控制信号配置**:
  - `ALUSrcA = 0` (或选择 PC 通路): 选择 PC 作为基准地址。
  - `ALUSrcB = 10`: 选择符号扩展并左移后的立即数 `ExtImm`。
  - `ALUOp`: 执行加法操作。
  - `PCWrite`: 在执行周期有效，将计算出的 BTA 直接写回 PC，完成跳转。

3.3 核心信号定义

- **CondEx (Condition Execute)**: 条件逻辑单元的核心输出。
  - 如果当前状态寄存器 CPSR 的标志位 (Flags) 满足指令中的条件码 ($Cond$)，则 $CondEx = 1$。
  - 否则 $CondEx = 0$。
- **FSM 原始信号**: $RegW$ (写寄存器请求), $MemW$ (写内存请求), $FlagW$ (写标志位请求), PCS (PC 更新请求), $NextPC$ (取指 PC 更新)。

3.4 时序流程分析 (Timing Analysis)

由于多周期处理器的状态随时间变化，必须确保控制信号在正确的时钟周期 (CLK) 生效，且避免逻辑竞争。

3.4.1 CLK1: 取指阶段 (Fetch)

- **状态**：S0 (Fetch)
- **操作**：$PC \leftarrow PC + 4$
- **逻辑**：FSM 输出 $NextPC = 1$。
- **结果**：根据逻辑公式 $PCWrite = 1 \vee (\dots) = 1$，PC 寄存器无条件更新。此时不关心 $CondEx$ 的值。

3.4.2 CLK2: 译码阶段 (Decode)

- **状态**：S1 (Decode)
- **操作**：指令译码，生成 $CondEx$。
- **逻辑**：指令已从内存读出，条件码 $Cond$ 已知。条件逻辑单元将其与旧的 $Flags$ 进行比较。
- **结果**：$CondEx$ **信号在此周期生成并稳定**，准备供后续阶段使用。

3.4.3 CLK3: 执行阶段 (Execute)

- **状态**：S2, S6, S7, S9 (Branch)
- **操作**：ALU 运算或分支跳转。
- **逻辑**：
  - 对于分支指令 (B)，如果 $CondEx = 1$，则 $PCS \wedge CondEx = 1$，触发 $PCWrite$，执行跳转。
  - 对于运算指令 (DP)，ALU 产生新的 $ALUFlags$ (N, Z, C, V)，但此时 CPSR 寄存器**尚未更新**。

3.4.4 CLK4: 写回/访存阶段 (Writeback / Mem)

这是时序最关键的阶段，存在标志位更新与写操作的潜在冲突。

- **关键冲突**：
  1. **写操作**：$RegW$ 或 $MemW$ 有效。此时 $CondEx$ 仍基于**旧的 Flags** (这是正确的，当前指令应基于旧状态执行)。
  2. **Flags 更新**：如果是带 S 的指令 (如 `ADDS`)，CPSR 会在 CLK4 的上升沿更新为新值。
- **为何安全？**

  虽然 CPSR 更新会导致 $CondEx$ 在 CLK4 周期内发生变化，但此时**当前指令的写操作已经完成** (或正在完成)。
- **CLK4+1 (下一周期)**：即使 $CondEx$ 因为新的 Flags 变成了 1，此时 FSM 已经跳转到 S0 (Fetch) 状态，控制信号 $RegW, MemW, PCS$ 全部变回 0。

$$RegWrite = 0 \wedge 1 = 0$$

因此，不会发生错误的写入。

3.4.5 CLK5: 存储器写回 (MemWB - LDR 指令)

- **状态**：S4
- **逻辑**：LDR 指令不更新 Flags ($FlagW = 00$)。因此 $CondEx$ 保持稳定，正确控制数据写入寄存器。

3.5 总结

多周期条件逻辑的设计精髓在于：

1. **条件预判**：$CondEx$ 在译码阶段即产生。
2. **信号互斥**：利用 FSM 的状态切换清除写使能信号，防止因 Flags 更新导致的 $CondEx$ 翻转引发误操作。
3. **精准门控**：通过 AND/OR 组合逻辑，精确区分了“系统必须执行的操作”(取指) 和“指令条件执行的操作”(写回/跳转)。

3.6 多周期处理器性能

处理器的性能核心指标是程序执行时间 ($ExecutionTime$)，其计算公式为：

$$\text{Execution Time} = \text{Instruction Count} \times \text{CPI} \times T_{cycle} \tag{12}$$

3.6.1 CPI (每指令周期数) 分析

多周期架构允许不同指令消耗不同数量的时钟周期。

指令周期分布 各指令的周期数及基准程序 (SPECINT2000) 的混合比例如下：

- **Branch (3 周期)**:
- **Data Processing / STR (4 周期)**:
- **LDR (5 周期)**:

平均 CPI 计算 加权平均 CPI 为：

$$\text{Average CPI} = (x \times 3) + (y \times 4) + (z \times 5) \tag{13}$$

3.6.2 时钟周期 (T_{c2}) 计算

多周期的时钟周期由最耗时的单个步骤（Stage）决定，而非整条指令。

关键路径公式 每个时钟周期都需要包含寄存器的时序开销和逻辑延迟：

$$T_{c2} = t_{pcq} + 2 \cdot t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \tag{14}$$

其中，max 函数体现了时钟频率受限于最慢的功能单元（通常是存储器访问）。

数值计算 使用 PPT 中提供的参数：

- $t_{pcq} = 40ps, t_{mux} = 25ps, t_{mem} = 200ps, t_{setup} = 50ps$

$$T_{c2} = 40 + 2(25) + 200 + 50 = \mathbf{340}\ \text{ps} \tag{15}$$

3.6.3 总体性能与对比

执行时间计算 假设程序包含 100×10^9 条指令：

$$\text{Time}_{multi} = (100 \times 10^9) \times 4.12 \times 340\ \text{ps} = \mathbf{140}\ \text{秒} \tag{16}$$

与单周期处理器对比

表 5: 单周期 vs 多周期性能对比

架构	CPI	时钟周期 (T_c)	总执行时间
单周期 (Single-Cycle)	1	840 ps	**84 s**
多周期 (Multicycle)	4.12	340 ps	**140 s**

结果分析 在本例中，多周期处理器的性能低于单周期处理器，主要原因如下：

1. **寄存器开销累积**：多周期每条指令平均需要经历 4.12 次寄存器建立与传输延迟 ($t_{setup} + t_{pcq}$)，而单周期仅需 1 次。
2. **步骤粒度不均**：时钟周期必须迁就最慢的存储器操作 (200ps)，导致较快的 ALU 操作 (120ps) 存在时间浪费。

尽管如此，多周期处理器在**硬件成本**（资源复用）上具有优势。

4 pipeline

4.1 流水线定义

自行看阅 PPT，笔者不想写了

4.2 数据冲突 (Data Hazard) 定义

数据冲突发生在当一条指令依赖于另一条尚未完成指令的结果时 [cite: 3993]。在五级流水线中，指令直到**写回阶段 (WB)** 结束时才将结果写入寄存器堆，但后续指令可能在**执行阶段 (EX)** 就需要读取该数据。如果硬件不采取措施，后续指令将会读取到旧的、错误的数据。

4.2.1 解决方法一：数据转发 (Data Forwarding)

4.2.2 原理

数据转发（或称旁路，Bypassing）通过硬件直接将计算结果从后级流水线（Memory 或 Writeback 阶段）传回前级（Execute 阶段）的 ALU 输入端，而无需等待数据写入寄存器堆 [cite: 3997]。

具体示例：RAW (Read After Write) 依赖 考虑以下汇编代码序列 [cite: 3994]：

```
1 ADD R1, R2, R3     ; 指令 1: R1 = R2 + R3
2 AND R4, R1, R5     ; 指令 2: R4 = R1 AND R5 (依赖 R1)
```

Listing 1: 数据转发示例代码

- **冲突描述**:
  - **指令 1 (ADD)**: 在周期 3 (EX) 计算出 $R1$ 的值，但直到周期 5 (WB) 才会写入寄存器堆。
  - **指令 2 (AND)**: 紧随其后，在周期 3 (ID) 读取寄存器，此时寄存器堆中的 $R1$ 仍是旧值。它需要在周期 4 (EX) 使用新的 $R1$。
- **硬件解决过程**:
  1. 在周期 4，指令 2 处于 **Execute (EX)** 阶段，指令 1 处于 **Memory (MEM)** 阶段。
  2. 转发单元检测到指令 2 的源寄存器 ($RA1E = R1$) 与指令 1 的目标寄存器 ($WA3M = R1$) 匹配，且指令 1 写使能有效 ($RegWriteM = 1$)[cite: 4001]。
  3. 控制逻辑设置多路选择器信号 `ForwardAE = 10`。
  4. ALU 的源操作数 A 直接从 **ALUResultM** (指令 1 的计算结果) 获取，而不是从寄存器堆读取。

4.2.3 解决方法二：流水线停顿 (Stalling)

4.2.4 原理

并非所有冲突都能通过转发解决。最典型的例子是**加载-使用冲突 (Load-Use Hazard)**。当一条指令需要使用上一条 `LDR` 指令从内存读取的数据时，数据直到访存阶段结束才能获得，这对于下一条指令的执行阶段来说太晚了，无法及时转发 [cite: 4005]。此时必须暂停流水线。

具体示例：LDR 引起的停顿 考虑以下代码 [cite: 4005]:

```
1 LDR R1, [R2, #10] ; 指令 1: 从内存加载数据到 R1
2 AND R4, R1, R5    ; 指令 2: 使用 R1 (Load-Use 冲突)
```

Listing 2: 流水线停顿示例代码

- **冲突描述**:
  - **指令 1 (LDR)**: 在周期 4 (MEM) 结束时才能从数据存储器读出数据。
  - **指令 2 (AND)**: 在周期 3 (EX) 就需要 $R1$ 的值进行 ALU 运算。
  - **问题**: 数据在周期 4 产生，但需要在周期 3 使用。这被称为“时间旅行”问题，无法通过简单的转发解决。
- **硬件解决过程 (Stall)**:

1. 在周期 3 (指令 2 的 Decode 阶段)，停顿单元检测到冲突：上一条指令是 `LDR` ($MemtoRegD\_EX = 1$) 且目标寄存器与当前源寄存器冲突。
2. **冻结 (Freeze)**：强制 $StallF = 1$ 和 $StallD = 1$。PC 和 Decode 流水线寄存器保持不变，指令 2 停留在译码阶段，指令 3 (未画出) 停留在取指阶段 [cite: 4006]。
3. **气泡 (Flush)**：强制 $FlushE = 1$。将执行级流水线寄存器清零。这意味着在下一个周期，执行阶段将执行一个 **NOP (空操作)**，而不是指令 2。
4. **结果**：流水线暂停一个周期。指令 1 进入 WB 阶段，数据准备好。指令 2 重新进入 EX 阶段，此时可以通过转发逻辑从 WB 阶段获取数据。

4.3 控制冲突定义

控制冲突（或称分支冲突）发生在流水线无法在取指阶段确定下一条指令的正确地址时。

- **原因**：分支指令（如 `B`）的目标地址通常在流水线较后的阶段（如执行级或写回级）才能计算出来。
- **后果**：如果流水线默认顺序取指，当分支发生跳转时，紧随其后进入流水线的指令就是错误的。这些错误指令必须被清除（Flush），从而导致性能损失。

4.3.1 场景一：分支指令 (Branch Instructions)

分支误预测 (Branch Misprediction) 如果处理器直到存储器访问阶段 (Memory Stage) 才决定是否跳转，所有的后续 4 条指令（F, D, E, M 阶段）都需要被清除。这被称为**分支误预测惩罚**。

优化方案：早期分支判断 (Early BTA) 为了减少惩罚，现代设计通常将分支判断和目标地址计算提前到 **执行阶段 (Execute Stage)**。

1. 硬件操作

- **地址计算**：ALU 在执行级计算分支目标地址 ($BTA = PC + Offset$)。
- **条件判断**：同时检查条件码（如 `EQ`, `NE`）决定是否跳转。
- **信号生成**：产生 `BranchTakenE` 信号。

2. 流水线冲刷 (Flush) 如果 `BranchTakenE` 为真（需要跳转），硬件必须在下一个时钟周期执行以下操作：

1. **更新 PC**：将 ALU 计算出的目标地址 `ALUResultE` 写回 PC。
2. **清除指令**：
   - **FlushD**：清除译码级寄存器（丢弃刚刚取到的指令）。
   - **FlushE**：清除执行级寄存器（丢弃刚刚译码的指令）。

结果：分支延迟减少到 2 个周期。

4.3.2 场景二：PC 写指令 (PC-Write Instructions)

除了标准的 `B` 指令，ARM 还允许通过数据处理指令直接修改 R15 (PC)，这也会引发控制冲突。

典型示例

```
ADD R15, R0, R1    ; 相当于跳转到 (R0 + R1) 的地址
MOV PC, R14        ; 子程序返回 (Return)
```

处理机制 (Stall + Flush) 这类指令与普通分支的区别在于，它们通常需要等待写回阶段才能真正更新 PC。文档指出这种冲突不能通过简单的数据转发来完全解决，因为目标是**取指地址**的改变，而不是数据的计算。

- **检测**：在译码阶段 (Decode)，检测到目标寄存器是 R15 (PC)。
- **动作**：
  1. **停顿 (Stall)**：如果需要，等待源寄存器数据准备好。
  2. **执行与写回**：指令流向执行级和写回级。
  3. **更新 PC**：当 `PCWrite` 信号在写回级生效时，PC 被更新。
  4. **冲刷 (Flush)**：由于 PC 更新较晚，流水线中已经预取了多条错误指令，这些指令所在的阶段（Fetch, Decode 等）必须被 Flush 掉。

4.4 流水线性能

4.4.1 CPI 计算

1. **Load 指令 (LDR)** (占比 25%)：
   - 无冲突 (60%)：1 周期

- 发生 Load-Use 冲突 (40%)：需停顿 1 周期 → 2 周期

$$CPI_{ldr} = 1 \times 0.6 + 2 \times 0.4 = 1.4 \tag{17}$$

2. **Branch 指令 (B)** (占比 13%)：

- 预测正确 (50%)：1 周期
- 预测错误 (50%)：罚时 2 周期 → 3 周期

$$CPI_{branch} = 1 \times 0.5 + 3 \times 0.5 = 2.0 \tag{18}$$

3. **Store (STR) & Data Processing (DP)** (占比 10% + 52%)：

- 无冲突：1 周期

$$CPI_{str} = CPI_{dp} = 1.0 \tag{19}$$

平均 CPI (Average CPI) 计算：

$$\begin{aligned} CPI_{avg} &= \sum(\text{Percentage}_i \times CPI_i) \\ &= (0.25 \times 1.4) + (0.10 \times 1.0) + (0.13 \times 2.0) + (0.52 \times 1.0) \\ &= 0.35 + 0.10 + 0.26 + 0.52 \\ &= \mathbf{1.23} \end{aligned} \tag{20}$$

4.4.2 时钟周期 (T_{c3}) 计算

流水线的时钟周期受限于最慢的一级。在本设计中，关键路径在于寄存器堆的读写（前半周期写，后半周期读），因此限制因素为寄存器读时间加上建立时间的两倍

$$\begin{aligned} T_{c3} &= 2 \times (t_{RFread} + t_{setup}) \\ &= 2 \times (100 + 50)\ \text{ps} \\ &= \mathbf{300}\ \text{ps} \end{aligned} \tag{21}$$

4.4.3 总时间计算

同前单周期，多周期 (终于编完了，崩溃了)