8 - Processor: Control

8.1 Identified Control Signals

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / **MemWrite**	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

8.2 Generating Control Signals: Idea

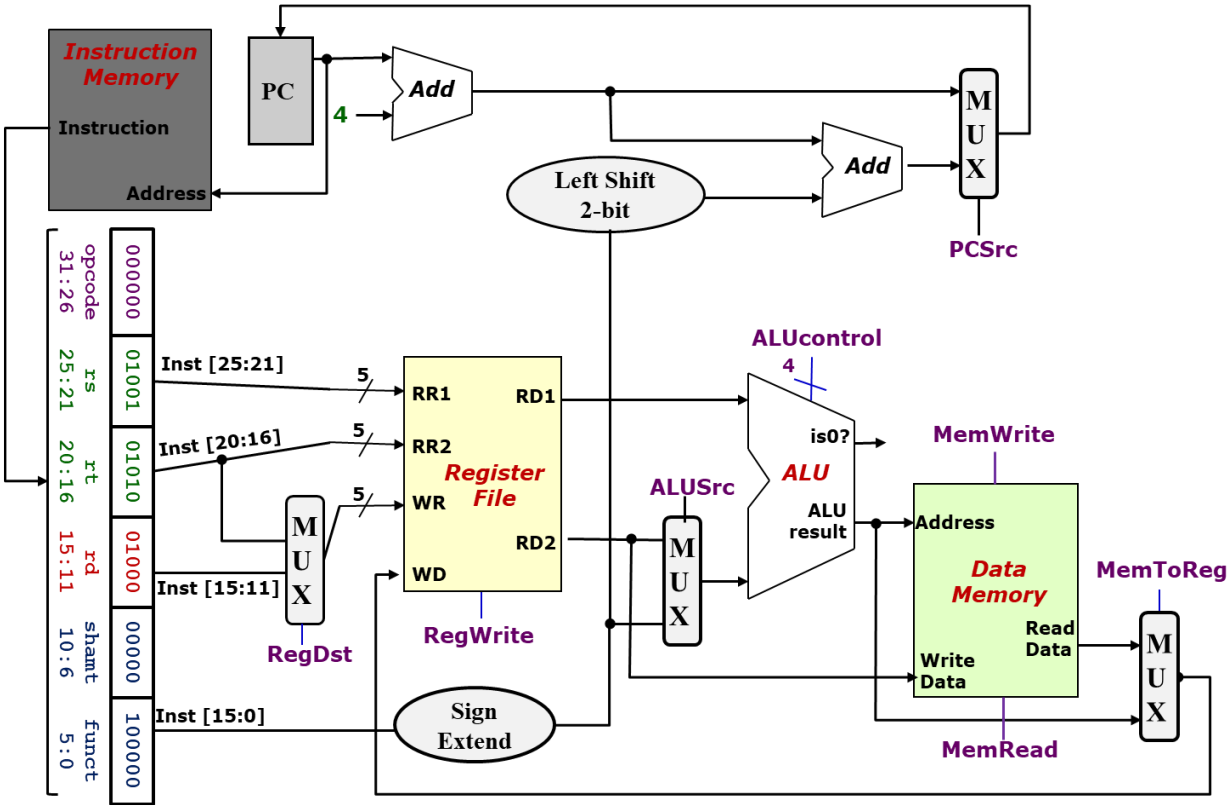
- The control signals are generated based on the instruction to be executed:
  - `opcode` -> Instruction Format
  - Example:
    - R-Format instruction -> `RegDst` = 1 (use `Inst[15:11]`)
  - R-Type instruction has additional information:
    - The 6-bit `funct` (function code, `Inst[5:0]`) field
- Idea:
  - Design a combinatorial circuit to generate these signals based on Opcode and possibly Function code
    - A control unit is needed

> 控制信号
>
> 1. 控制信号的生成：
>   - 控制信号是基于要执行的指令而生成的。这些信号告诉数据路径硬件如何执行指令。例如应该执行哪种算术或逻辑操作，数据应该来自哪里已经结果应该存储在哪里
> 2. `opcode`

- 所有MIPS指令的开始部分都有一个操作码(opcode)，它决定了指令的基本操作类型。通过解码(decode)这个操作码，可以知道要执行的指令类型，从而生成相应的控制信号

3. 指令格式与 `RegDst`
   - 例如对于R-Format（寄存器格式）的指令，有一个控制信号 `RegDst` 决定目标寄存器的选择。如果在R-Format指令中 `RegDst` 设置为1，则意味着目标寄存器的信息来自于 `Inst[15:11]` 字段
4. **R-Type指令的额外信息**：
   - R-Type指令除了操作码外，还有一个6位的函数代码（funct）字段，即 `Inst[5:0]` 。这个函数代码进一步指定了R-Type指令的具体操作，例如加法、减法等。
5. **主要思想**：
   - 设计一个组合逻辑电路，根据指令的操作码（Opcode）和可能的函数代码（Function code）生成这些控制信号。
   - 为了生成和管理这些控制信号，需要一个控制单元。

8.3 The Control Units

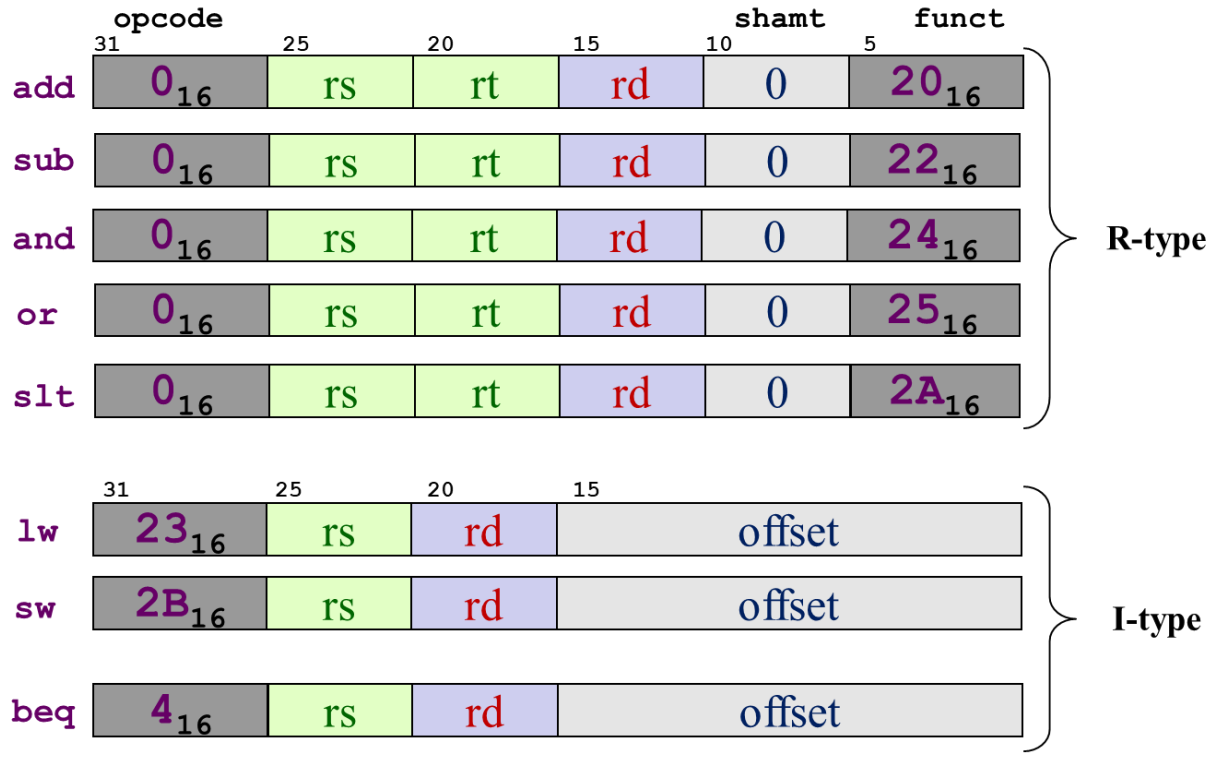


8.3.1 Implement the Control Unit

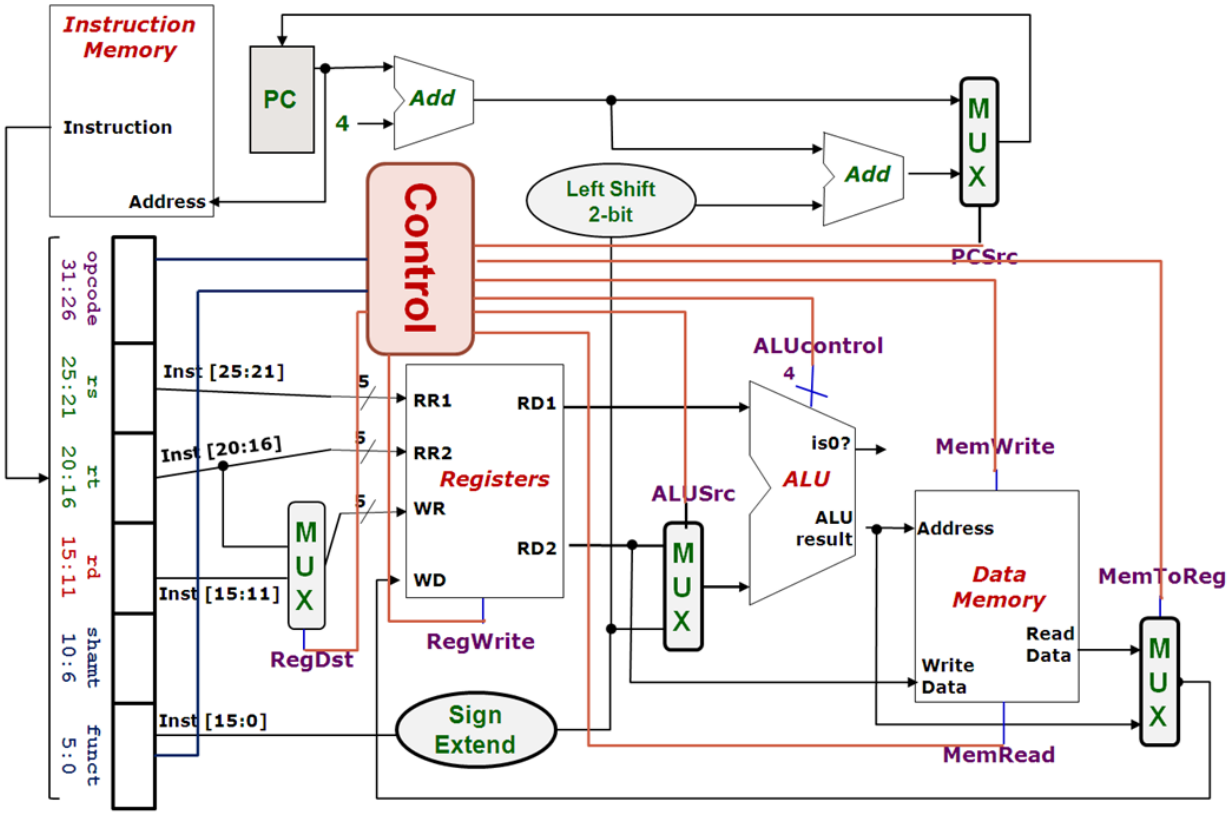
- Approach:
  - Take note of the instruction subset to be implemented:
    - `opcode` and function code
  - Go through each signal:
    - Observe how the signal is generated based on the instruction opcode and/or function code
  - Construct truth table

- Design the control unit using logic gates

8.3.2 MIPS Instruction Subset



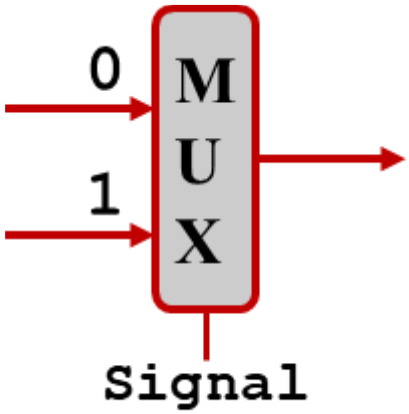
8.4 Control Signals



8.4.1 RegDst

> 这个控制信号决定了要写入的目标寄存器

- False (0): Write register = `Inst[20:16]`
- True (1): Write register = `Inst[15:11]`



8.4.2 RegWrite

> 决定是否将新的值写入寄存器

- False (0): No register write
- True (1): New value will be written

8.4.3 ALUSrc

> 这个信号决定了ALU（算术逻辑单元）的第二个操作数来源

- False (0): Operand2 = Register Read Data 2
- True (1): Operand2 = SignExt(Inst[15:0])

8.4.4 MemRead

> 决定是否执行内存读取操作

- False (0): Not performing memory read access
- True (1): Read memory using `address`

8.4.5 MemWrite

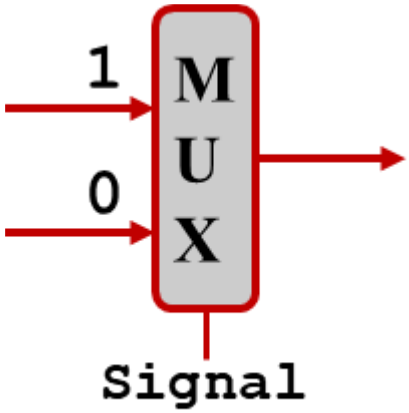
> 决定是否执行内存写入操作

- False (0): Not performing memory write operation
- True (1): memory[address] <- Register Read Data 2

8.4.6 MemToReg

> 这个信号决定了要写入寄存器的数据来源

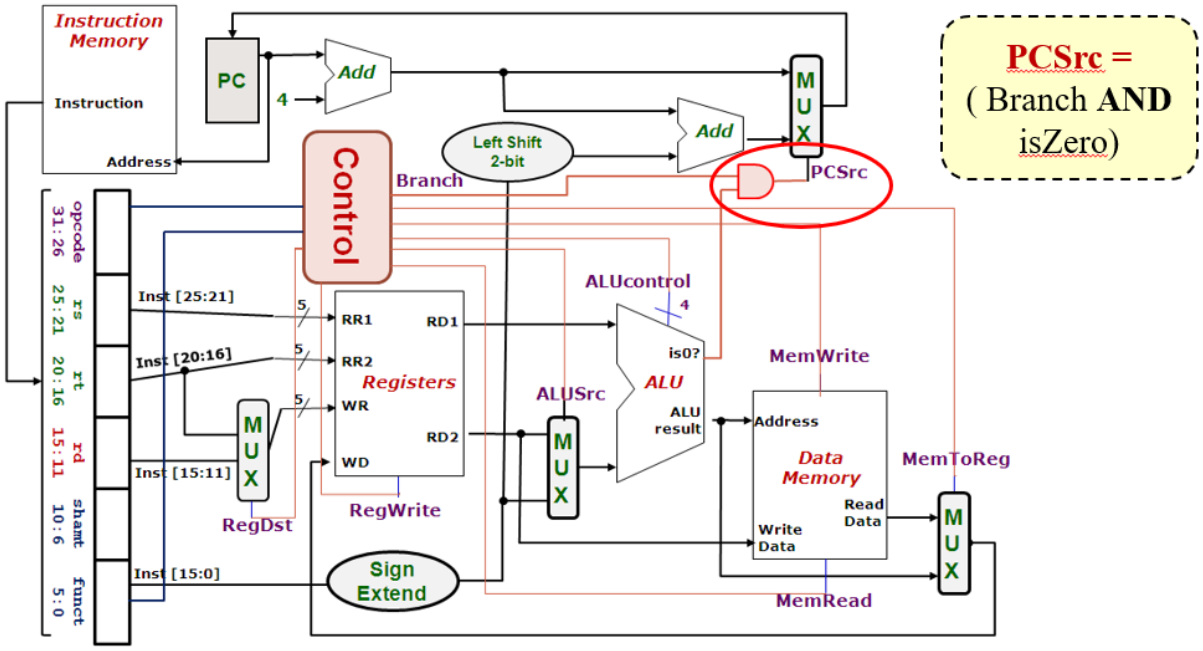
- True (1): Register write data = Memory read data
- False (0): Register write data = ALU result
- Important: The input of MUX is swapped in this case



8.4.7 PCSrc

> 这个控制信号基于ALU的 `isZero` 信号来确定分支指令的实际结果（是否执行分支）

- The `isZero` signal from the ALU gives us the actual branch outcome (taken/not taken)
- Idea: “If instruction is a branch AND taken, then…”
- False (0): Next PC = PC + 4
- True (0): Next PC = SignExt(Inst[15:0]) << 2 + (PC + 4)



Summary

Observation so far:

- The signals discussed so far can be generated by `opcode` directly
  - Function code is not needed up to this point
- A major part of the controller can be built based on `opcode` alone

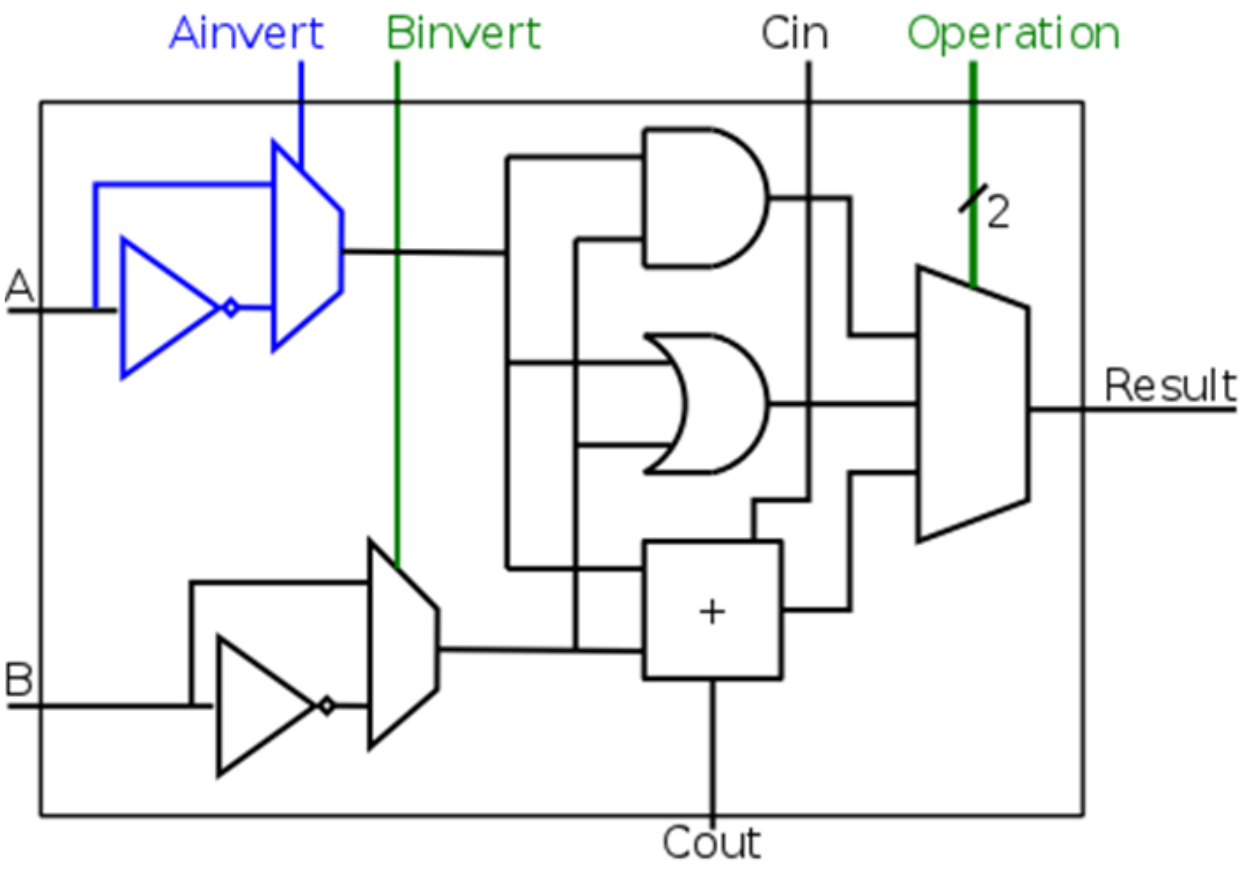
8.5 ALU Control Signal

- The ALU is a combinatorial circuit:
  - Capable of performing several arithmetic operations

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

8.5.1 One Bit At A Time

- A simplified 1-bit MIPS ALU can be implemented as follows:



- 4 control bits are needed:
  - `Ainvert`
    - 1 to invert input A
  - `Binvert`
    - 1 to invert input B
  - `Operation` (2-bit)
    - To select one of the 3 results

ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

ALU（算术逻辑单元）通常执行多种算术和逻辑操作，而这些操作是通过内部的控制信号激活的。在某些设计中，这些控制信号中的一部分可能包括Ainvert、Binvert和Operation。

1. **Ainvert**：此信号用于控制是否应该对输入A进行求反（即位反转或数值取反）。当Ainvert设置为1时，ALU会对A的所有位进行反转（例如，从二进制的"0"变为"1"，反之亦然）。这在某些操作（例如减法）中是有用的，因为它们可以通过使用加法器和求反逻辑来简化。
2. **Binvert**：与Ainvert类似，Binvert控制是否对输入B进行反转。这也是实现减法等操作的常用技巧，因为通过求反和加法，可以很容易地在已有的硬件上执行减法。
3. **Operation**：这是一个2位的字段，它直接定义了ALU应执行的操作类型。由于它是一个2位信号，因此它可以表示4种不同的操作（例如，00表示加法，01表示减法，10表示AND操作，11表示OR操作等）。实际的操作和编码会根据具体的ALU设计而变化。

现在，让我们看看这些信号是如何协同工作来控制ALU的：

- **实现减法**：要使用ALU执行减法，我们可以利用加法器硬件来执行该操作。理论上，A - B 可以重写为 A + (-B)。因此，我们可以设置Ainvert为0（保持A不变），Binvert为1（求B的二进制反码），然后通过Operation信号告诉ALU执行加法操作。通常，还需要在B的反码上加1（即取补码），以完成从正数到负数的转换。
- **实现逻辑操作**：对于逻辑操作（如AND、OR、NOR等），Ainvert和Binvert通常会设置为0，这样A和B就保持不变。相应的操作是通过Operation字段的2位代码来指定的，这会直接控制ALU内部执行哪种逻辑操作。

这些信号的组合允许ALU利用较少的硬件资源（主要是加法器和逻辑单元）来执行一系列的算术和逻辑操作。通过巧妙地利用位反转和选择不同的操作类型，ALU可以用相对简单的方式实现复杂的功能。

8.5.2 Multilevel Decoding

- Now we can start to design for `ALUcontrol` signal, which depends on:
  - `opcode` (6-bit) field and `Function Code` (6-bit) field
- Brute Force approach
  - Use `opcode` and `function code` directly, i.e. finding expressions with 12 variables
- Multilevel Decoding approach:
  - Use some of the input to reduce the cases, then generate the full output
  - Simplify the design process, reduce the size of the main controller, potentially speedup the circuit

> `ALUcontrol` 信号的生成取决于指令的两个字段： `opcode` （操作码，6位）和 `Function Code` （功能码，也是6位）。这些字段确定了CPU需要执行的具体操作。
>
> 1. **蛮力方法 (Brute Force approach)**：
>   - 这种方法直接使用 `opcode` 和 `function code` ，即通过寻找包含12个变量的表达式来生成 `ALUcontrol` 信号。这相当于直接对所有可能的输入组合进行硬编码，非常直接但可能会非常复杂，因为它需要处理所有的 `opcode` 和 `function code` 组合。
> 2. **多级解码方法 (Multilevel Decoding approach)**：
>   - 这种方法更加巧妙。它首先使用部分输入（比如只用 `opcode` 字段）来减少需要直接解码的情况数量。基于这个初步的解码，控制逻辑可以将可能的操作范围缩小到更易管理的数量。
>   - 然后，系统可能会根据需要考虑 `Function Code` 来进一步确定要执行的确切操作，从而生成完整的 `ALUcontrol` 信号。这样做简化了设计过程，因为不是每个操作都需要单独编码，同时还减小了主控制器的大小。
>   - 由于解码器不必同时处理所有的12个变量，这种方法还可能加快电路的速度。处理更少的变量意味着更快的逻辑运算，从而可能提高整个处理单元的响应时间。
>
> **多级解码 (Multilevel Decoding)** 的概念是通过在多个阶段处理输入信息来减少同时处理的变量数量，简化电路设计，提高解码效率。在第一级，解码器可能只考虑一部分输入变量并做出部分决策；在随后的级别，它会逐步考虑更多的变量，逐渐缩小操作的范围。这样做的好处是可以简化每个阶段的逻辑复杂性，减少所需硬件的数量，并提高操作速度。

8.5.3 Intermediate Signal: `ALUop`

- Basic Idea:
  1. Use `opcode` to generate a 2-bit `ALUop` signal
     - Represents classification of the instructions

Instruction type	ALUop
lw / sw	00
beq	01
R-type	10

  2. Use `ALUop` signal and `function code` field to generate the 4-bit `ALUcontrol` signal

引入一个中间信号 ALUop 来简化控制信号的生成。这是多级解码策略的一个实例，其目的是减少复杂性并提高系统效率。以下是这个过程的详细解释：

基本思路：

1. **使用 opcode 生成2位的 ALUop 信号**：
   - 在这个阶段，系统读取指令的 opcode （操作码），这是指令中的一个字段，表示要执行的操作的类型（例如，加载、存储、分支、算术运算等）。然后，这个 opcode 被解码为一个更简单的2位信号 ALUop ，它表示指令的分类。不同的 opcode 将导致不同的 ALUop 信号。
   - 例如，表中列出了三种类型的指令（ lw / sw ， beq ，和R类型），每种类型都被分配了一个特定的 ALUop 代码。
2. **使用 ALUop 信号和 function code 字段生成4位的 ALUcontrol 信号**：
   - 接下来， ALUop 信号和指令中的 function code （功能码）一起用于确定确切的操作，该操作应由ALU执行。
   - function code 是指令的另一个部分，仅在某些类型的指令（如R类型）中使用，它提供了关于应执行的确切算术或逻辑操作的更多信息。
   - 根据 ALUop 和 function code 的组合，生成一个4位的 ALUcontrol 信号，该信号直接控制ALU，告诉它要执行的确切操作（例如，加、减、与、或等）。

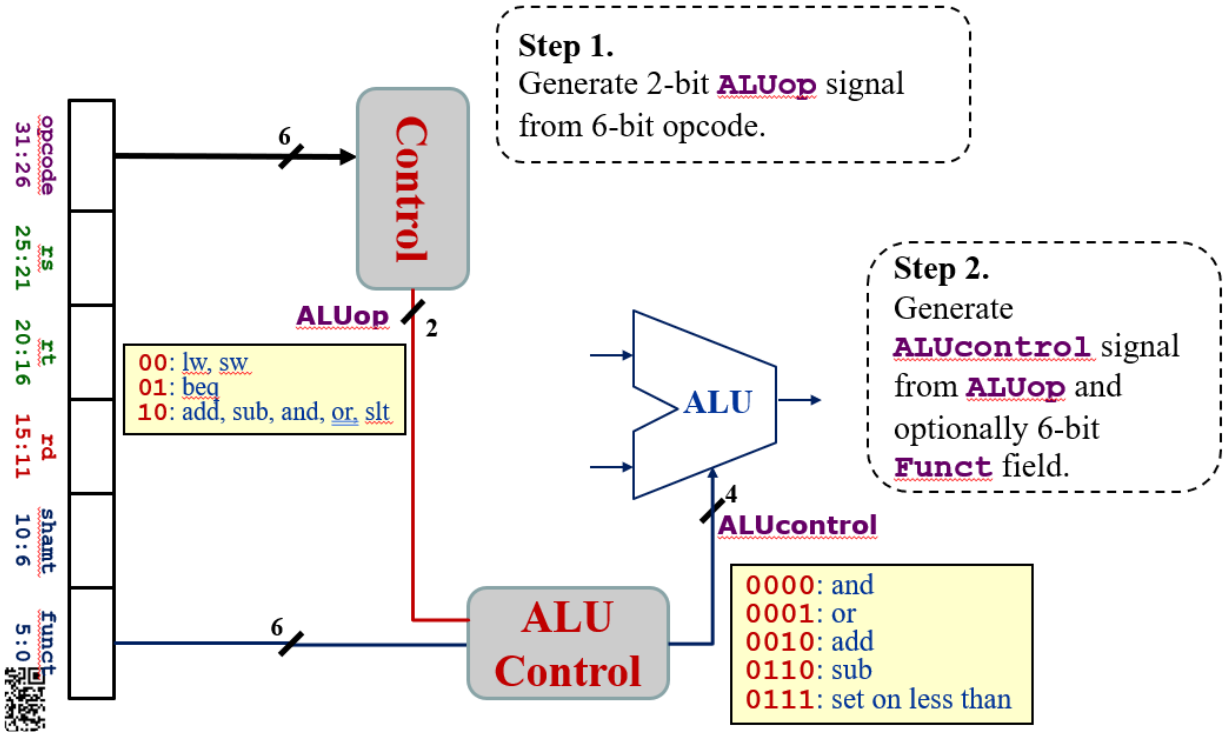
为什么需要 ALUop ：

引入 ALUop 作为一个中间步骤有几个好处：

1. **简化解码**：通过首先将 opcode 转换为一个更简单的 ALUop 信号，解码器可以在处理完整的 opcode 和 function code 之前，先进行一次“预解码”，这减少了同时需要考虑的变量数量。
2. **减少硬件复杂性**：直接解析整个 opcode 和 function code 可能需要很多逻辑门，而这种方法通过减少每个阶段需要的逻辑复杂性来减少所需的硬件。
3. **模块化设计**：这种分级方法允许设计者在不同的层次上考虑问题，可能使得测试和故障排除更加容易。

通过这种分步骤的方法，系统可以更有效地生成正确的 ALUcontrol 信号，即使在面对多种可能的操作和复杂的指令集时也是如此。这就是多级解码在实际应用中的一个例子。

8.5.4 Two-Level Implementation



8.5.5 Generating `ALUcontrol` Signal

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

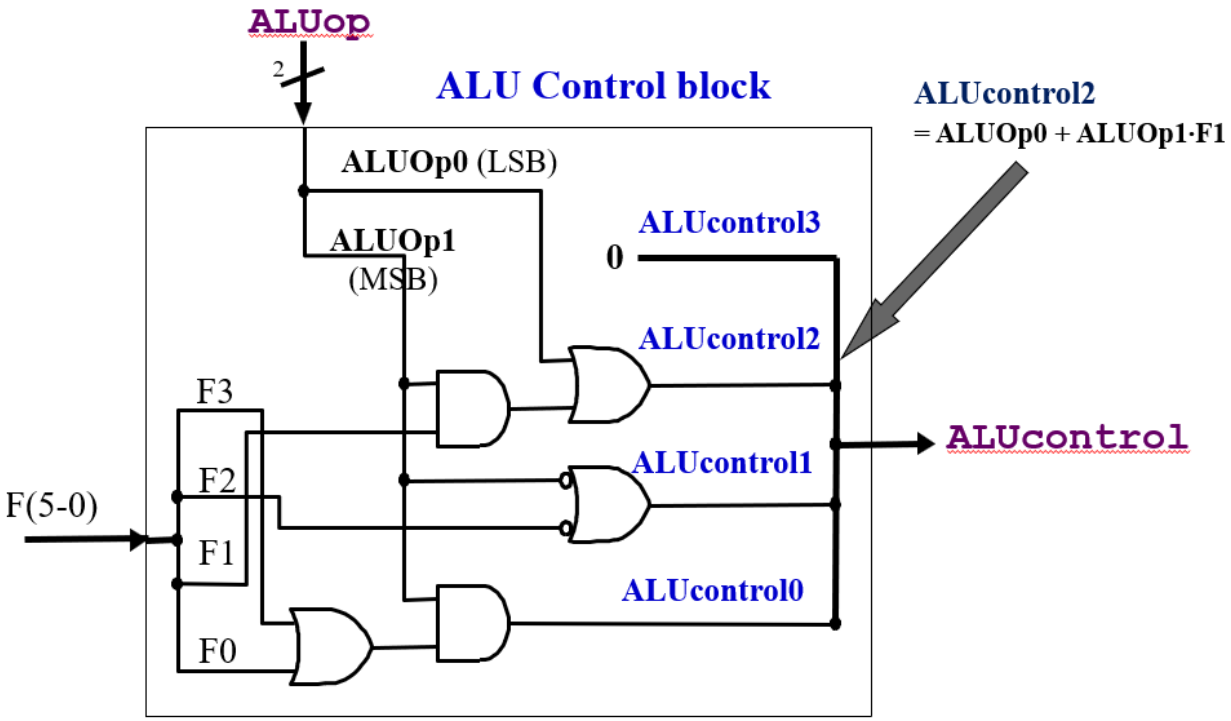
Generation of 2-bit `ALUop` signal will be discussed later

8.5.6 Design of ALU Control Unit

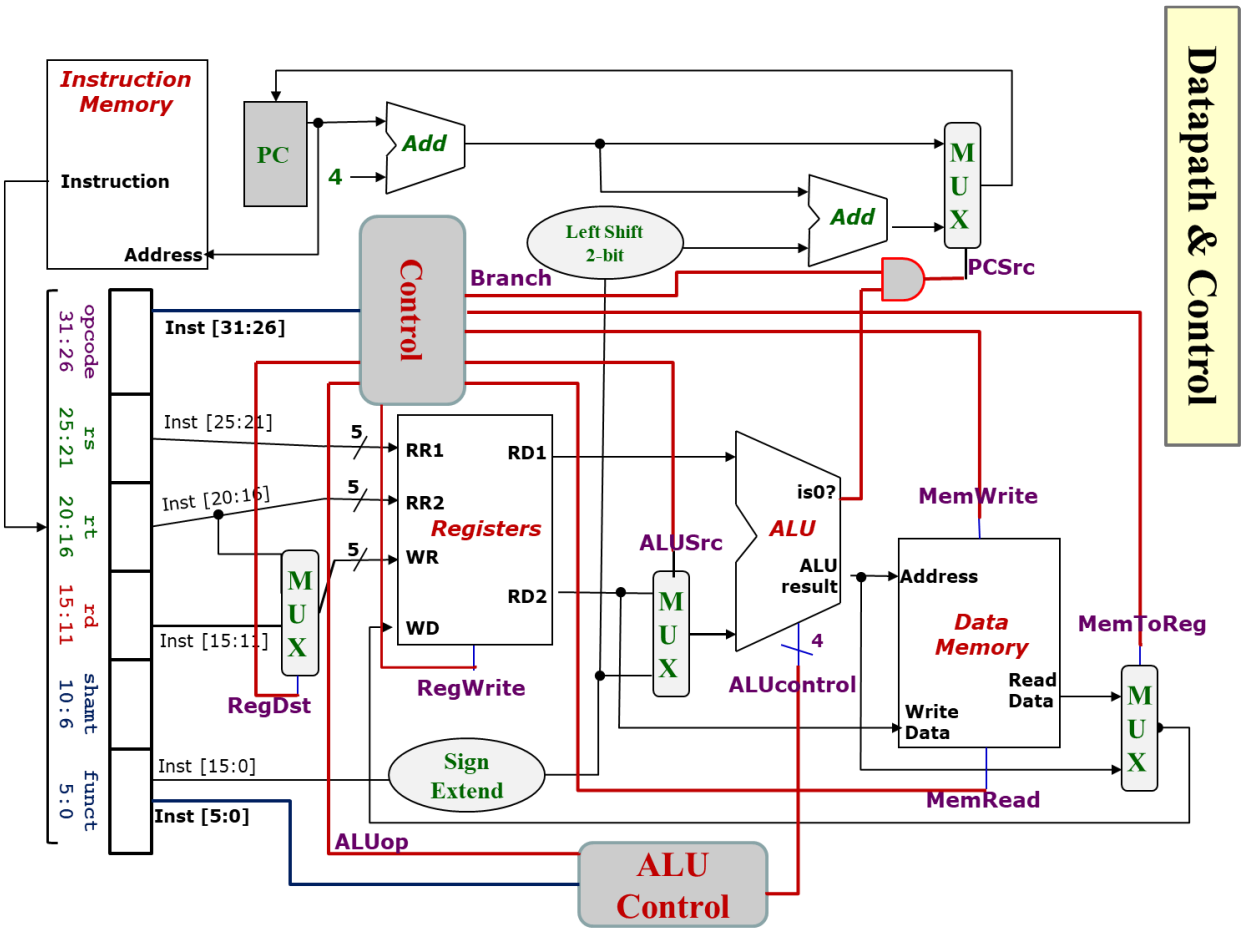
- Input: 6-bit `funct` field and 2-bit `ALUop`
- Output: 4-bit `ALUcontrol`

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 1 1 1

- Simple combinational logic



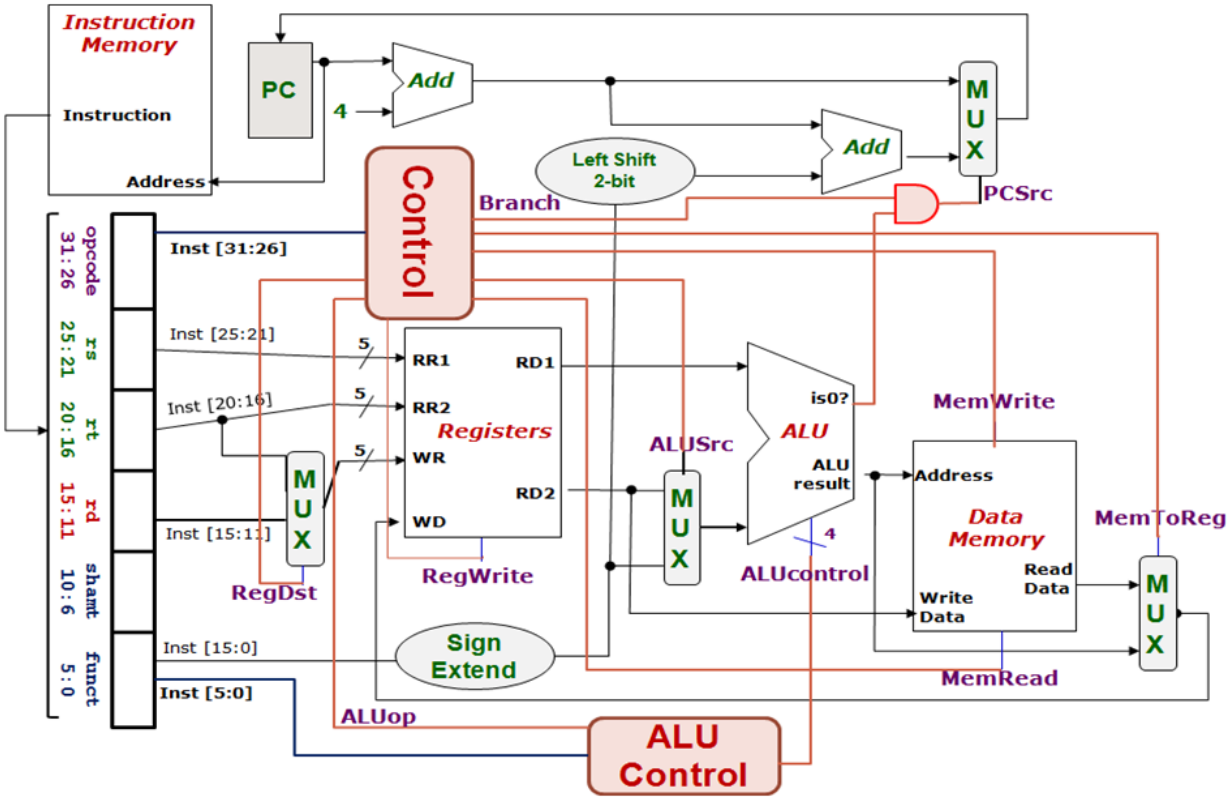
8.5.7 Summary



- Typical digital design steps:
  - Fill in truth table
    - Input: `opcode`
    - Output: Various control signals as discussed
  - Derive simplified expression for each signal

8.5.8 Control Design: Outputs

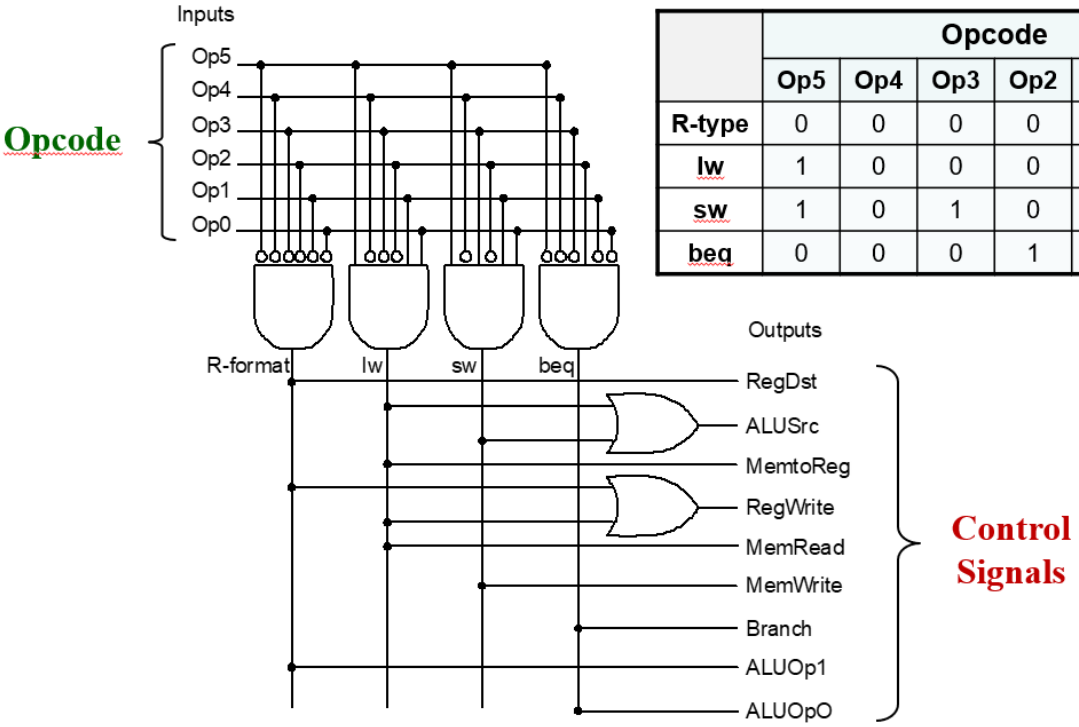
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



8.5.9 Control Design: Inputs

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

8.5.10 Combinational Circuit Implementation



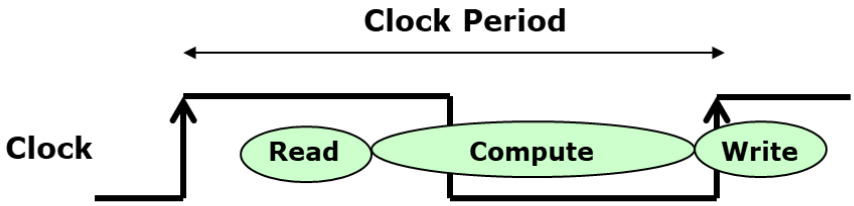
	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

8.6 Instruction Execution

Instruction Execution =

1. Read contents of one or more storage elements (register/memory)
2. Perform computation through some combinational logic
3. Write results to one or more storage elements (register/memory)

All these performed within a clock period



Don’t want to read a storage element when it is being written.

8.6.1 Single Cycle Implementation: Shortcoming

Calculate cycle time assuming negligible delays: memory (2ns), ALU/adders (2ns), register file access (1ns)

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	**6**
lw	2	1	2	2	1	**8**
sw	2	1	2	2		**7**
beq	2	1	2			**5**

All instructions take as much time as the slowest one (i.e. load)

-> Long cycle time for each instruction

> 这段内容在讨论单周期处理器实现的一个重要限制。在单周期（Single Cycle）处理器架构中，处理器在一个时钟周期内完成整个指令。时钟周期的长度由执行所有指令所需的最长路径决定。这里提供了一个简化的例子来说明这个概念。
>
> 首先，表格列出了不同类型指令在各个阶段所需的时间。这些阶段包括指令内存访问（从内存中获取指令）、寄存器文件读取（从寄存器读取数据）、ALU操作（执行算术或逻辑操作）、数据内存访问（对内存进行加载或存储操作）和寄存器写入（将结果写回寄存器）。每个阶段的延迟被假设为一个确定的值，例如内存访问为2纳秒，ALU和加法器操作为2纳秒，寄存器文件访问为1纳秒。
>
> 现在，考虑到不同指令的需求，表中展示了它们各自所需的总时间。例如，“lw”（加载字）指令需要8纳秒，因为它涉及所有的步骤：指令内存、寄存器读取、ALU操作、数据内存和寄存器写入。
>
> 然而，单周期处理器的一个关键缺点是所有的指令必须在一个单一的、固定长度的时钟周期内完成。这个周期的长度由最慢的指令（在这个例子中是“lw”指令，需要8纳秒）决定。即使其他指令（如“beq”或ALU操作）可以更快地完成，时钟周期也不能更短，因为它必须足够长以容纳最慢的指令。结果是，所有的指令都会受到最慢指令的“拖累”，导致整体性能的下降。
>
> 总结一下，单周期实现的主要缺点是：
>
> 1. 时钟周期时间由最慢的指令决定，导致效率低下。
> 2. 更快的指令不得不等待，不能立即释放系统资源，从而减少了处理器的吞吐量。
> 3. 不能充分利用可能的并行性，因为下一指令直到当前指令完成之后才开始，即使它所需的资源已经可用。
>
> 这些限制促使了其他类型的CPU设计，例如多周期和流水线架构，这些架构可以更有效地处理指令集中的时间差异。

8.6.2 Solution #1: Multicycle Implementation

Break up the instructions into execution steps:

1. Instruction fetch
2. Instruction decode and register read
3. ALU operation
4. Memory read/write
5. Register write

Each execution step takes one clock cycle

- Cycle time is much shorter, i.e., clock frequency is much higher

Instructions take variable number of clock cycles to complete execution

8.6.3 Pipelining

Break up the instructions into execution steps one per clock cycle

Allow different instructions to be in different execution steps simultaneously

> “Pipelining”是计算机架构中提高处理器性能的关键技术之一。它不是通过“切片”来实现的，而是通过将指令处理过程分解为几个连续的步骤或阶段，每个步骤在各自的硬件中执行。这些步骤或阶段是按照顺序排列的，每个阶段完成一个特定的部分操作。通过这种方式，处理器可以在不同阶段同时处理多条指令。
>
> **实现流水线的步骤通常包括：**
>
> 1. **指令取回（Instruction Fetch，IF）** - 处理器从内存中读取下一条要执行的指令。
> 2. **指令译码（Instruction Decode，ID）** - 解码器将二进制指令解码为处理器可以理解的指令，并确定需要使用的数据。
> 3. **执行（Execute，EX）** - ALU（算术逻辑单元）执行所需的计算，比如加法、减法、乘法等。
> 4. **内存访问（Memory Access，MEM）** - 如果指令需要，处理器会在这一步读取或写入数据到内存。
> 5. **写回（Write-back，WB）** - 处理器将执行结果写回到寄存器。
>
> 在流水线处理中，上述每个步骤都在各自的时钟周期内发生，并且它们是重叠的。例如，在一个给定的时钟周期内，一条指令可能处于“执行”阶段，而另一条指令可能处于“指令取回”阶段。这就允许在每个时钟周期内启动一条新的指令，大大提高了处理器的吞吐量和效率。
>
> 这种方法的效率很大程度上依赖于指令和流水线阶段的划分能否使得每个阶段都尽可能短且均衡，从而避免某个阶段过长而造成的瓶颈。