Chapter 4 — The Processor

4.1 Introduction

4.2 Logic Design Conventions

4.3 The Single-Cycle Design

4.4 The Pipelined Design

4.1 Introduction

In this chapter we will examine in more detail the inner workings of a computing system, in particular, the CPU. Two separate designs for a simple MIPS32-like processor are discussed: the **single-cycle** design and a **pipelined** design.

A Basic MIPS Implementation

It would be impossible in the time we have allotted for Chapter 4 to discuss the design of a processor that would implement every MIPS32 instruction, so Chapter 4 focuses on a select subset of instructions to give you an idea of how R-, I-, and J-format instructions are executed. The subset of instructions are,

Memory reference instructions:	`lw`, `sw` (I-format)
Arithmetic-logical instructions:	`add`, `sub`, `and`, `or`, `nor`, `slt` (R-format)
Branch instructions:	`beq` (I-format) , `j` (J-format)

4.2 Logic Design Conventions

A microprocessor is a complex digital circuit which executes instructions. It can be divided into two primary parts: the **datapath** and the **control unit** (or simply, the **control**).

The **datapath** consists of those elements that store data (bits), move bits around, and operate on bits. Datapath elements include,

Digital logic gates (AND, OR, NAND, XOR, etc)
Muxes, Decoders, Encoders
Adders, Multipliers, Dividers, Shifting circuits
Register file (CPU registers)
Instruction and data memory
Cache memory

The Arithmetic Logic Unit (ALU) is a major component of the datapath and contains the circuitry for performing arithmetic operations (`+`, `-`, `*`, `/`), as well as other operations, e.g., logical AND.

The **control** is responsible for generating **control signals** which control the behavior of the datapath. There are two common techniques for implementing the control,

Hardwiring - employs sequential logic and a finite state machine (FSM) to generate the control signals.
Microprogramming - microinstructions (called microcode) are executed to generate the control signals.

Each has its advantages and disadvantages (control is discussed in more detail in Appendix D). In Chapter 4, a simple hardwired control unit is designed.

Note that the wall time it takes to execute an instruction will vary depending on the instruction. For example, a `lw` may take 50 ns (memory accesses are relatively slow) whereas `add` may require 100 ps (assuming the operands are already in registers and available to be sent to the ALU).

It is more difficult to design a microprocessor circuit in which instructions execute in a variable amount of time. For this reason, in Chapter 4, a **single-cycle** design is implemented first. In the single-cycle design, every instruction takes one clock cycle to execute—even though each individual instruction will take a variable amount of time within that clock cycle.

Question: If each instruction requires variable time to complete, how do we ensure that each instruction completes within one clock cycle?

Answer: Make the clock period greater than or equal to the time required for the slowest instruction. For example, if `lw` is the slowest instruction, at 50 ns, then make the clock period $>$ 50 ns (or the clock frequency $<$ 200 MHz).

I hope you will notice that the single-cycle is very inefficient, but understand for learning purposes, we are more interested in the simplicity of the design than the performance.

The single-cycle design implements a **Harvard architecture**[1]: separate instruction and data memories. The primary advantage is that both memories can be simultaneously accessed. The main disadvantage is that it requires two buses.

1 The term originated with the Harvard Mark I (officially the IBM Automatic Sequence Controlled Calculator or ASCC) designed and built by IBM in 1944.

Some small microcontrollers and DSP controllers—used primarily in embedded systems—implement a Harvard architecture. Most other processors, and certainly the type found in desktops, laptops, pad and cell phones, implement a Princeton architecture or **Von Neumann architecture**[2]: one combined memory for both instructions and data. (Looking ahead to Chapter 5, most modern systems actually implement a modified Harvard architecture where there are separate instruction and data caches backed by a common memory storing both instructions and data.)

4.2.1 Clocking

In digital logic, components employ **combinational** or **sequential** logic.

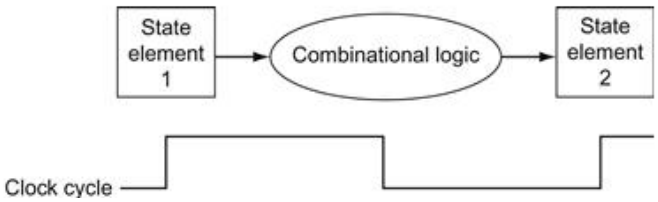
A combinational logic component is one where the outputs depend only on the inputs, e.g., a logic gate.

A sequential logic component is one that contains **state** (i.e., internal storage or memory) and where the outputs depend on both the current inputs and the state of the component.

All modern processors are designed using **synchronous** digital logic, i.e., a **clock** is involved. The clock is used to determine when data are valid and stable (i.e., the signal on the wire has reached a steady 0 or 1 state) relative to the clock, and to control when writes to sequential logic components occur.

In the clocking methodology, events can be based on the **level** of the clock signal (low or high) and in this case, the events are said to be **level-triggered**. Alternatively, events can be based on the **edge** of the clock signal (rising or falling) and these events would be **edge-triggered**. Edge-triggering is more common and is used in Chapter 4.

For an example of how edge-triggering is used see Fig. 4.3 in the textbook.



During a single-clock cycle, this sequence of events will occur,

> Signals emanating from State Element 1 are "read" and fed into the combinational logic.
> The combinational logic will perform some operation which will take x amount of time.
> On the next rising clock edge, combinational logic outputs will be written to State Element 2.

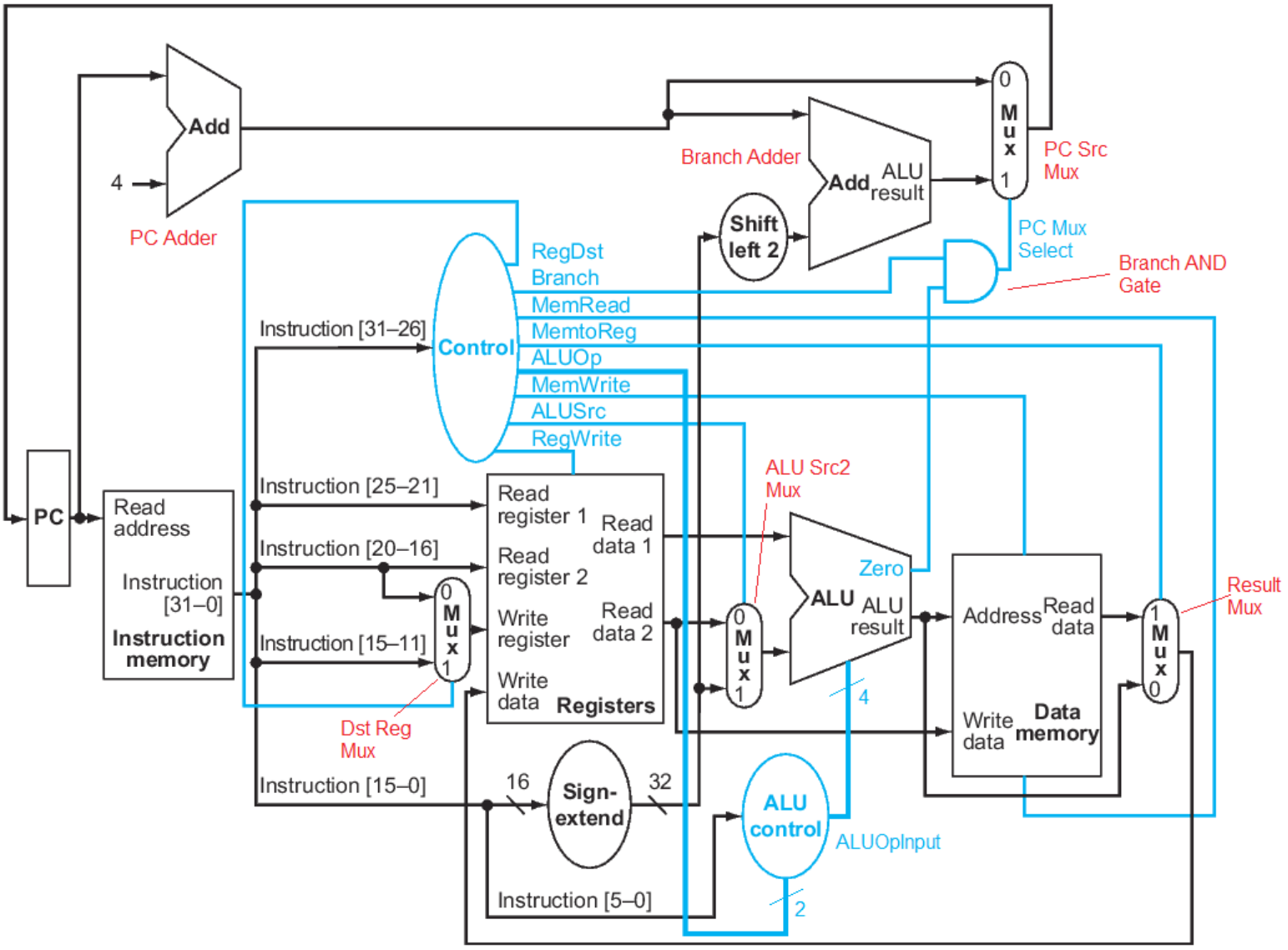
The clock period must be greater than or equal to x, i.e., the combinational logic must have enough time to generate its results prior to the next clock edge, but note that the prior contents of State Element 2 (i.e., its state) will be coming out of State Element 2 just ahead of the bits being written on the rising clock edge. State Element 2 will perform some internal logic on the inputs, taking y amount of time, which will change its state. This may change its outputs, so the outputs of State Element 2 will not be stable until at least y time units after the rising edge of the clock. Consequently the **clock cycle time** (or clock period) must be greater than or equal to $x + y$, and since **clock frequency** is inversely related to clock period, the clock frequency must be less than or equal to 1 over $x + y$.

2 The term originated in a 1945 draft document describing the design of a computer system known as EDVAC (Electronic Discrete Variable Automatic Computer) which was authored by John Von Neumann, a renowned physicist and mathematician at the Princeton Institute for Advanced Study.

Note: In the diagrams of Chapter 4, state elements are assumed to be written at the end of every clock cycle, which is also the same moment in time as the beginning of the next clock cycle. If a state element is not written at the end of every clock cycle, a separate **write control signal** must be connected to it so that it can be written when required.

4.3 The Single-Cycle Design

Fig. 4.17 shows the datapath and control that implements the subset of instructions mentioned in §4.1, with the exception of the `j` instruction not yet being implemented (we will see how `j` is implemented shortly). I have augmented the diagram in the book by assigning names to some of the datapath components.



4.3.1 The Machine Cycle

The digital logic circuit comprising the processor implements what we will refer to as the **machine cycle** (other terms include **instruction cycle** and **fetch-decode-execute cycle**). The machine cycle begins with the rising edge of the system clock.

1. At the rising edge of the clock, the 32-bit instruction *Instr* is fetched from InstructionMemory[PC].
2. In parallel do:
   a. PC Adder: Compute PC + 4
   b. Send instruction opcode bits $Instr_{31:26}$ to control unit for decoding
   c. Send instruction function code bits $Instr_{5:0}$ to ALU control unit for decoding.
   d. Send instruction bits $Instr_{25:21}$ for register *rs* operand to register file.
   e. Send instruction bits $Instr_{20:16}$ for register *rt* operand to register file.
   f. Send either instruction bits $Instr_{20:16}$ or $Instr_{15:11}$ for register *rt* or *rd* operand to register file.
   g. Control unit decodes opcode bits $Instr_{31:26}$ to determine which instruction this is:
      1. If the instruction is `lw` then the ALU Control configures the ALU to perform an addition; the sum is sent as the address to DataMemory; the data word is read; the word is sent to the register file for writing to *rt* on the next clock edge.
      2. If the instruction is `sw` then the ALU Control configures the ALU to perform an addition; the sum is sent as the address to DataMemory; the word read from the source register *rt* is written into DataMemory on the next clock edge.
      3. If the instruction is an arithmetic-logical instruction (R-format), then the two operands *rs* and *rt* are read from the register file and sent to the ALU; the control unit sends a 2-bit signal to the ALU Control to help it determine which operation to perform; the ALU Control sends a 4-bit signal to the ALU telling it which operation to perform; the ALU result is sent back to the register file for writing to *rd* on the next clock edge.
      4. If the instruction is `beq`, then the two operands *rs* and *rt* are sent from the register file to the ALU; the ALU Control configures the ALU to perform a subtraction; the ALU will assert the Zero control signal if $rs = rt$; the 16-bit immediate value in $Instr_{15:0}$ is sign-extended to a 32-bit immediate value; the Branch Adder computes the branch target address; if Zero is asserted the PC Src Mux will be selected so the branch target address is written to PC on the next clock edge.
      5. What happens during a `j` instruction is discussed in §4.3.13.
      6. If the instruction is not `beq` or `j` then the PC Src Mux is selected so PC+4 is written to PC on the next clock edge.
3. On the next rising edge of the clock, any writes to state elements, e.g., registers, take place at the same time that Step 1 starts again. This cycle is repeated millions, billions, trillions, or gazillions of times per second.

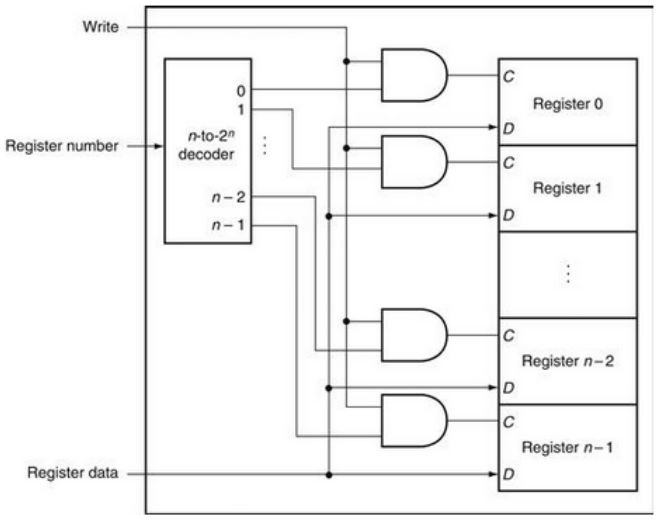
4.3.2 Datapath Components: Program Counter Register

The **Program Counter** (PC) register always contains the address in the **Instruction Memory** (IMEM) of the instruction that will be fetched and executed in the next clock cycle.

Assuming no branch or jump is taken, the address in IMEM of the instruction that will be executed next is PC + 4 which is calculated by the **PC Adder**.

4.3.3 Datapath Components: Register File

The processor's 32 general-purpose registers $0 through $31 are stored in a structure termed the **Register File**. Fig. 8.9 in Appendix B (shown below) shows one way to construct an n-register register file. Each 32-bit register could be constructed using 32 D flip-flops and would have two inputs: C is a CLK signal that is asserted to write to the register and D is the 32-bit word to write.



In the single-cycle datapath we write to a register by performing these steps,

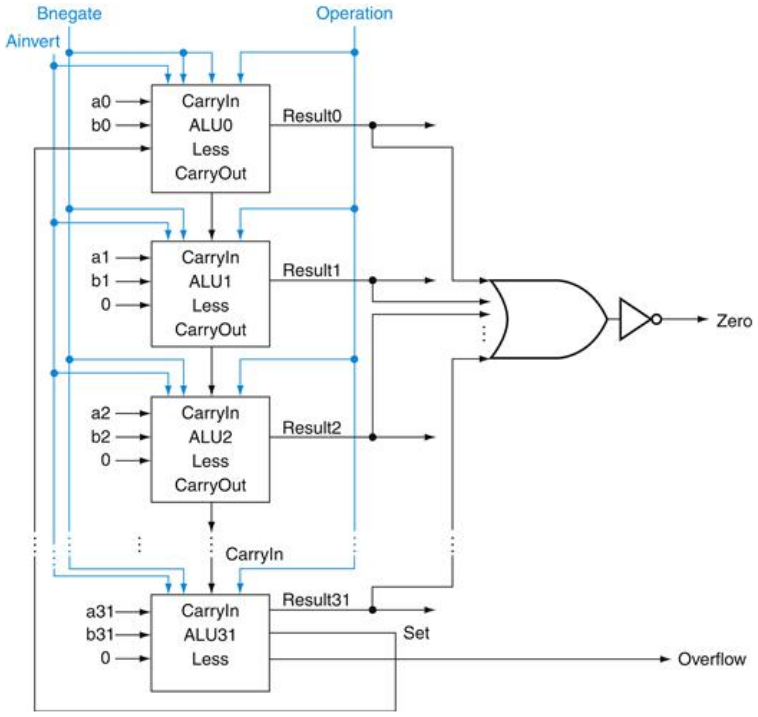
1. Put the 5-bit register number on the *Write Register* input to the register file.
2. Put the 32-bit word to be written on the *Write Data* input.
3. Activate the *RegWrite* control signal to cause the write to occur on the next rising clock edge.

Internal to the register file, the 5-to-32 decoder selects the appropriate register for writing by ensuring that the output of only one AND gate will be asserted. The other input to each AND gate is the *RegWrite* control signal which is asserted at the end (beginning) of each clock cycle when a write must occur.

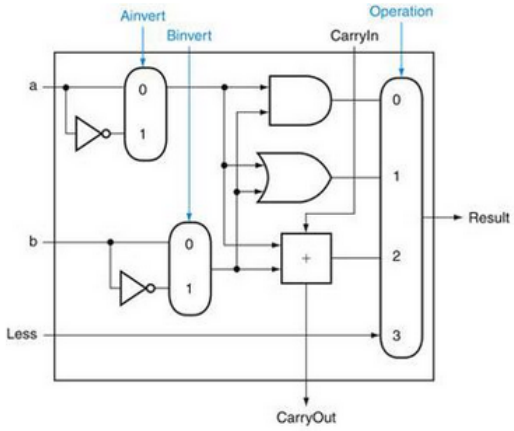
4.3.4 Datapath Components: ALU

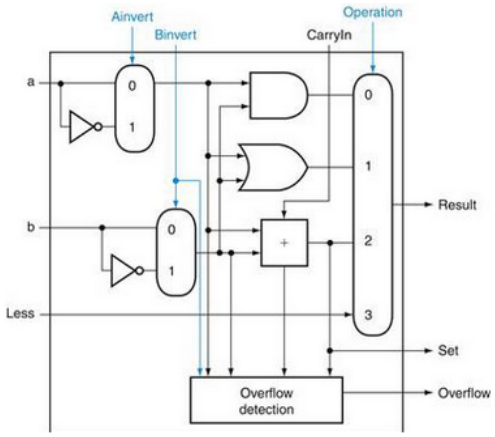
The construction of a 32-bit ALU is discussed in §B.5 of Appendix B. The ALU implements the following operations on 32-bit inputs A and B,

$A \wedge B$	Logical AND
$A \vee B$	Logical OR
$A + B$	Addition
$A - B$	Subtraction
SLT	Set on less than (1 when $A < B$, 0 otherwise)
$\overline{A \vee B}$	Logical NOR



The 32-bit ALU is constructed from 32 1-bit ALU's (ALU_0 through ALU_{31}) with ALU_0 through ALU_{30} being implemented using the 1-bit ALU shown below left and ALU_{31} being constructed from the 1-bit ALU shown below right.





Each 1-bit ALU contains,

1. Four 1-bit data inputs,

a	One bit of the full 32-bit ALU operand *A*.
b	One bit of the full 32-bit ALU operand *B*.
CarryIn	The carry input signal from the previous mini-ALU for the internal full adder.
Less	Used in implementing the SLT instruction.

2. Three control inputs,

Ainvert	When asserted, $\overline{a}$ is used in the resulting operation rather than *a*.
Binvert	When asserted, $\overline{b}$ is used in the resulting operation rather than *b*.
Operation	A 2-bit signal that selects the mux that outputs the *Result*.

3. ALU_0 through ALU_{30} have two data outputs,

Result	The 1-bit result of performing the operation selected by *Operation.*
CarryOut	The carry out bit from the internal full adder.

4. ALU_{31} has three data outputs,

Result	The 1-bit result of performing the operation selected by *Operation.*
Set	Used in implementing the `slt` instruction.
Overflow	Asserted high when an arithmetic operation results in overflow.

The 32-bit ALU has three control inputs,

Ainvert	Used in computing $\overline{A \vee B}$
Bnegate	Used in computing $\overline{A \vee B}$ and in implementing subtraction
Operation	A 2-bit signal that selects each of the 4-to-1 muxes in ALU_0 through ALU_{31}.

For ALU_0 through ALU_{31}, the 2-bit *Operation* signal controls one of four operations by selecting the output of the 4-to-1 *Result* mux. In conjunction with *Ainvert* and *Binvert* this permits the ALU to perform these operations,

Ainvert	***Binvert***	***Operation***	***Result***	**Implements**
0	0	00	$a \wedge b$	Logical AND
0	0	01	$a \vee b$	Logical OR
1	1	00	$\overline{a} \wedge \overline{b} = \overline{a \vee b}$	Logical NOR
0	0	10	sum bit from $a + b$	Addition
0	1	10	sum bit from $a + \overline{b}$	Subtraction
0	1	11	*Less* input	SLT

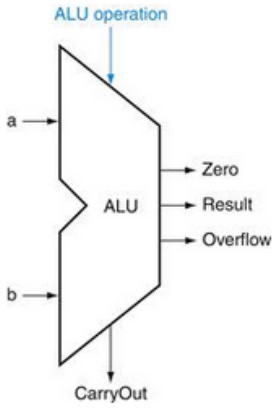
Subtraction of B from A is performed as: $A - B = A + \text{-}B$. The negation of B in two's complement is accomplished by forming the one's complement of B (by inverting each b bit) and adding 1. The addition of 1 is accomplished by connecting $Operation_1$ (the msb) to *CarryIn* of ALU_0. Since *Operation* will be 10_2 for subtraction, this sets *CarryIn* to 1.

The `slt rd, rs, rt` instruction is performed by subtracting B (register *rt*) from A (register *rs*). If A is less than B then the result of the subtraction will be a negative value; if A is not less than B then the result of the subtraction will be zero or positive. In two's complement, a negative 32-bit integer has bit 31 set and a nonnegative integer has bit 31 cleared. Bit 31 of our 32-bit result will be set when the sum bit from the full adder of ALU_{31} is 1 and bit 31 will be cleared when the sum bit of this full adder is 0. Therefore, the sum bit from the full adder of ALU_{31} forms the *Set* output of ALU31 and this output is fed back to become the *Less* input of ALU_0. Consequently, when *Operation* = 11 the ALU *Result* will be 1 if A is less than B and 0 if A is greater than or equal to B.

ALU31 contains overflow detection logic which we will not discuss.

To implement `beq rs, rt, label` and `bne rs, rt, label` in hardware requires us to determine if $rs = rt$ and if $rs \neq rt$, respectively. We can easily determine if $rs = rt$ or if $rs \neq rt$ by detecting if *rs* - *rt* is zero (as it will be when $rs = rt$) or nonzero (as it will be when $rs \neq rt$). That is the function of the large 32-bit XNOR gate: when all of the *Result* bits are 0, the *Zero* control signal will be 1; if one or more *Result* bits are 1, then *Zero* will be 0. We will see how *Zero* is used later in implementing `beq` and `bne`.

The standard circuit symbol for the full 32-bit ALU is,



In the diagram for the datapath of the single-cycle design, the **ALU Control** logic block generates a 4-bit ALU control signal we shall label $ALUOpInput_{3:0}$ where,

$ALUOpInput_3$	Connects to *Ainvert*
$ALUOpInput_2$	Connects to *Bnegate*
$ALUOpInput_1$	Connects to internal $ALUOperation_1$
$ALUOpInput_0$	Connects to internal $ALUOperation_0$

Thus,

ALUOpInput	**ALU Operation**
0000	Logical AND
0001	Logical OR
0010	Addition
0110	Subtraction
0111	Set on less than
1100	Logical NOR

The **ALU Control** is controlled by two inputs: the 6-bit function code field from instruction bits $Instr_{5:0}$ and a 2-bit signal from the main control unit named $ALUOp_{1:0}$ which is encoded as,

`lw`	00 (I)	`sub`	10 (R)
`sw`	00 (I)	`and`	10 (R)
`beq`	01 (I)	`or`	10 (R)
`j`	xx (J)	`nor`	10 (R)
`add`	10 (R)	`slt`	10 (R)

For `lw` and `sw` the ALU needs to perform an addition. The 2-bit $ALUOp$ signal going from the Control to the ALU Control is set to 00 and the ALU control must output $ALUOpInput = 0010$.

For `beq` the ALU needs to perform an addition. The 2-bit $ALUOp$ signal going from the Control to the ALU Control is set to 01 and the ALU control must output $ALUOpInput = 0110$.

For `j`, we do not use the ALU, so the 2-bit $ALUOp$ signal can be set to anything, i.e., $ALUOp = xx$.

For R-format instructions—`add`, `sub`, `and`, `or`, `nor`, `slt`—the main control unit sets $ALUOp$ to 10 and the ALU control must output 0010, 0110, 0000, 0001, 1100, or 0111.

All of this can be summarized in a truth table where the inputs are $ALUOp_{1:0}$ and $Instr_{5:0}$ and the output is $ALUOpInput_{3:0}$.

Instruction	$ALUOp_{1:0}$	$Instr_{5:0}$	$ALUOpInput_{3:0}$	
`lw`	00	xxxxxx	0010	`lw`, `sw`, `beq`, and `j` do not have a function field
`sw`	00	xxxxxx	0010	so we do not look at $Instr_{5:0}$ when determining
`beq`	01	xxxxxx	0110	$ALUOpInput$.
`j`	xx	xxxxxx	xxxx	
`add`	10	xx0000	0010	All of these R-format instructions have 10_2 as
`sub`	10	xx0010	0110	the two msb's in $Instr_{5:0}$ so we only look at the
`and`	10	xx0100	0000	four lsb's when determining $ALUOpInput$.
`or`	10	xx0101	0001	
`nor`	10	xx0111	1100	
`slt`	10	xx1010	0111	

Drawing the resulting combinational logic block that this truth table produces will be left as an exercise to the student (truthfully, it is really not as complex as you might think it would be).

4.3.5 Datapath Components: ALU Src 2 Mux

The **ALU Src 2 Mux** is used to select the second operand for an ALU operation.

Instruction	ALU Source Operand 1	ALU Source Operand 2
`lw`, `sw`	*rs*	*sign-ext*($imm_{15:0}$)
`beq`	*rs*	*rt*
R-format	*rs*	*rt*
`j`	n/a	n/a

4.3.6 Datapath Components: Sign Extend Unit

We can always prepend 0-bits to a nonnegative binary number, changing the bit-length of the integer, without changing the number itself, e.g.,

$1011_2 = 11_{10} \rightarrow 000000001011_2 = 11_{10}$

The same is true of a negative binary number, i.e., we can always prepend 1-bits, e.g.,

$1111111111111010_2 = \text{-}5_{10} \rightarrow 11111111111111111111111111111010_2 = \text{-}5_{10}$

The **Sign Extend** unit is used to convert the 16-bit immediate encoded in I-format instructions to a 32-bit immediate by replicating the sign bit. There is no logic involved: this simply involves wiring,

4.3.7 Datapath Components: Shift Left by Two Unit

Shifting left by two is simply wiring as well,

4.3.8 Datapath Components: Branch Adder

The beq instruction is an I-format instruction with the 16-bit immediate field being used to compute the branch target address using this formula,

$$branch\ target\ address = (\text{PC} + 4) + sign\text{-}ext(imm_{15:0}) << 2 \qquad \text{Eqn. 1}$$

Most processor architectures form the branch target address by using an addressing mode which is known as **PC-relative addressing**, where an **offset** is added to PC to form the branch target address. If we form the 32-bit branch target address by adding PC and the 16-bit immediate field of the branch instruction (which forms a two's complement offset in the range [-32768, 32767]) then we could branch to any address which is in the range [PC - 32768, PC + 32767]. To extend this range, MIPS treats the offset as being words rather than bytes, i.e., $offset = sign\text{-}ext(imm_{15:0}) << 2$.

Although branch instructions are common—they are used to implement if statements and loops, which are very common in HLL programming—the majority of instructions that are executed are not branches. This means that the majority of the time, the next instruction that is executed will be the instruction following the one that is currently being executed. Consequently, the PC Adder proceeds to immediately calculate PC + 4 *before* the Control and the rest of the datapath have figured out that the instruction is a branch instruction and the branch will or will not be taken. Therefore, PC + 4, rather than PC, is the input to the Branch Adder.

Example: Consider this code. What would be the encoding of the beq and bne instructions assuming that the address of the add instruction is 0x0040_4000?

```
loop:
   add   $t0, $t0, $t1       # 0x0040_4000
   sll   $t0, $t0, 2         # 0x0040_4004
   slt   $t1, $t0, $zero     # 0x0040_4008
   beq   $t1, $zero, false   # 0x0040_400C
   li    $t2, 13             # 0x0040_4010
   j     end_if              # 0x0040_4014
false:
   li    $t2, -13            # 0x0040_4018
end_if:
   bne   $t0, $zero, loop    # 0x0040_401C
end_loop:
   nop                       # 0x0040_4020
```

When the `beq` instruction is fetched from memory to be executed, PC is 0x0040_400C and PC+4 is 0x0040_4010. The branch target address is 0x0040_4018, calculated using Eqn. 1. The assembler must write the branch offset to the *imm* field of the instruction encoding. To determine what the immediate value is, we can solve Eqn. 1 for $imm_{15:0}$,

$$imm_{15:0} = (branch\text{-}target\text{-}address_{31:0} - (PC + 4)) >> 2 \quad \text{Eqn. 2}$$

Consequently, for the `beq` at 0x0040_400C,

$imm_{15:0} = (0x0040\_4018 - (0x0040\_400C + 4)) >> 2 = (0x0040\_4018 - 0x0040\_4010) >> 2$
$imm_{15:0} = 0x8 >> 2$
$imm_{15:0} = 1000_2 >> 2$
$imm_{15:0} = 0000\ 0000\ 0000\ 0010_2$

The encoding for `beq` will be,

```
000100 01001 00000 0000000000000010 = 0x11200002
op     rs    rt    imm
```

When the `bne` instruction is fetched from memory to be executed, PC is 0x0040_401C and PC+4 is 0x0040_4020. The branch target address is 0x0040_4000. Consequently,

$imm_{15:0} = (0x0040\_4000 - (0x0040\_401C + 4)) >> 2 = (0x0040\_4000 - 0x0040\_4020) >> 2$
$imm_{15:0} = -0x20 >> 2$ (note: -0x20 in 32-bit two's complement is 0xFFFFFFE0)
$imm_{15:0} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0000 >> 2$
$imm_{15:0} = 1111\ 1111\ 1111\ 1000$ (which is -8 in decimal)

The encoding for `bne` will be:

```
000101 01000 00000 1111111111111000 = 0x1500FFF8
op     rs    rt    imm
```

4.3.9 Datapath Components: PC Src Mux

The value to be written to PC at the next clock is selected by what I will refer to as the **PC Src Mux**, which is selected by the output of an AND gate. The inputs to the AND gate are a control signal named *Branch* emanating from the Control (asserted when $Instr_{31:26}$ = 000100, beq, or $Instr_{31:26}$ = 000101, `bne`) and the *Zero* output from the ALU. Remember that *Zero* is 1 when the ALU result is 0. During the execution of `beq` the contents of registers *rs* and *rt* will be fed into the ALU and *ALUOpInput* will be configured to perform a subtraction. If *rs* = *rt* then the ALU result will be 0 and both *Zero* and *Branch* will be 1, thus selecting the output from the Branch Adder. If *rs* $\neq$ *rt* *Zero* will be 0 which causes the AND gate to output 0 which selects PC + 4 as the address to be written to PC. The `bne` instruction operates similarly.

4.3.10 Datapath Components: Dst Reg Mux

For R-format instructions (`add`, `sub`, `and`, `or`, `nor`, `slt`) at the next clock edge we must write a result to the *rd* register (encoded in $Instr_{15:11}$). `sw`, `beq`, and `j` do not write a result, but `lw` does, and for `lw`, the register to be written is in *rt* (encoded in $Instr_{20:16}$). Therefore, the **Dst Reg Mux** is used to select the destination register for those instructions that write to the register file. The mux is selected by a signal named *RegDst* which emanates from the Control.

4.3.11 Datapath Components: Result Mux

The **Result Mux** selects the word to be written to the destination register on the next clock edge. For R-format instructions, the word will be *ALU Result*, but for `lw` the value will be a word read from the **Data Memory** (DMEM).

4.3.12 The Control Unit

How to decode the instruction? Decoding is performed by examining the *op* bits in $Instr_{31:26}$ and for R-format instructions, the function bits in $Instr_{5:0}$.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

R-format (ADD, AND, NOR, OR, SUB, SLT)

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

I-format (LW, SW)

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

I-format (BEQ)

Thus,

Instruction	Format	Instr[31:26] Op	Instr[5:0] Funct
ADD	R	000 000	100 000
AND	R	000 000	100 100
BEQ	I	000 100	xxx xxx
J	J	000 010	xxx xxx
LW	I	100 011	xxx xxx
NOR	R	000 000	100 111
OR	R	000 000	100 101
SLT	R	000 000	101 010
SUB	R	000 000	100 010
SW	I	101 011	xxx xxx

The Control receives the opcode bits of the instruction, examines them, and using combinational logic, asserts control signals going to different parts of the datapath.

	ADD	AND	BEQ	J	LW	NOR	OR	SLT	SUB	SW
ALUOp[1:0]	10	10	01	xx	00	10	10	10	10	00
ALUSrc	0	0	0	x	1	0	0	0	0	1
Branch	0	0	1	x	0	0	0	0	0	0
MemRead	0	0	0	x	1	0	0	0	0	0
MemToReg	0	0	x	x	1	0	0	0	0	x
MemWrite	0	0	0	0	0	0	0	0	0	1
RegDst	1	1	x	x	0	1	1	1	1	x
RegWrite	1	1	0	0	1	1	1	1	1	0

Combining these two tables, we get the master control table, at the top of the next page.

Instr	Op[5]	Op[4]	Op[3]	Op[2]	Op[1]	Op[0]	Fn[5]	Fn[4]	Fn[3]	Fn[2]	Fn[1]	Fn[0]	ALUOp	ALUSrc	Branch	MemRead	MemToReg	MemWrite	RegDst	RegWrite	ALUOpInput
ADD	0	0	0	0	0	0	1	0	0	0	0	0	10	0	0	0	0	0	1	1	0010
AND	0	0	0	0	0	0	1	0	0	1	0	0	10	0	0	0	0	0	1	1	0000
BEQ	0	0	0	1	0	0	x	x	x	x	x	x	01	0	1	0	x	0	x	0	0110
J	0	0	0	0	1	0	x	x	x	x	x	x	xx	x	x	0	x	0	x	0	0000
LW	1	0	0	0	1	1	x	x	x	x	x	x	00	1	0	1	1	0	0	1	xxxx
NOR	0	0	0	0	0	0	1	0	0	1	1	1	10	0	0	0	0	0	1	1	1100
OR	0	0	0	0	0	0	1	0	0	1	0	1	10	0	0	0	0	0	1	1	0001
SLT	0	0	0	0	0	0	1	0	1	0	1	0	10	0	0	0	0	0	1	1	0111
SUB	0	0	0	0	0	0	1	0	0	0	1	0	10	0	0	0	0	0	1	1	0110
SW	1	0	1	0	1	1	x	x	x	x	x	x	00	1	0	0	x	1	x	0	0010

Drawing the resulting combinational logic is left as an exercise for the student.

4.3.13 Implementing Jump

Implementing `j` is straightforward. The formula to compute the jump target address is,

$$jump\ target\ address = (\text{PC}+4)_{31:28} \ || \ (Instr_{25:0} << 2) \qquad \text{Eqn. 3}$$

where || means bit-concatenation. All of this is accomplished with wiring, a new mux (**PC Src Mux 2**), and a new control signal named *Jump* that is asserted when $Instr_{31:26} = 000\ 010$.

Example: Consider this code. What would be the encoding of the `j` instruction assuming that the address of the `add` instruction is 0x0040_4000?

```
loop:
   add   $t0, $t0, $t1        # 0x0040_4000
   sll   $t0, $t0, 2          # 0x0040_4004
   slt   $t1, $t0, $zero      # 0x0040_4008
   beq   $t1, $zero, false    # 0x0040_400C
   li    $t2, 13              # 0x0040_4010
   j     end_if               # 0x0040_4014
false:
   li    $t2, -13             # 0x0040_4018
end_if:
   bne   $t0, $zero, loop     # 0x0040_401C
end_loop:
   nop                        # 0x0040_4020
```

When the `j end_if` instruction is fetched from memory to be executed, PC is 0x0040_4014 and PC+4 is 0x0040_4018. The jump target address is 0x0040_401C, calculated using Eqn. 3. The assembler must write part of the jump target address to the *addr* field of the instruction encoding. To determine what the *addr* value is, we can solve Eqn. 3 for $addr_{25:0}$,

$$addr_{25:0} = (jump\text{-}target\text{-}address_{31:0} - [(\text{PC}+4)_{31:28} \ || \ 0\text{x}0000000]) >> 2 \qquad \text{Eqn. 4}$$

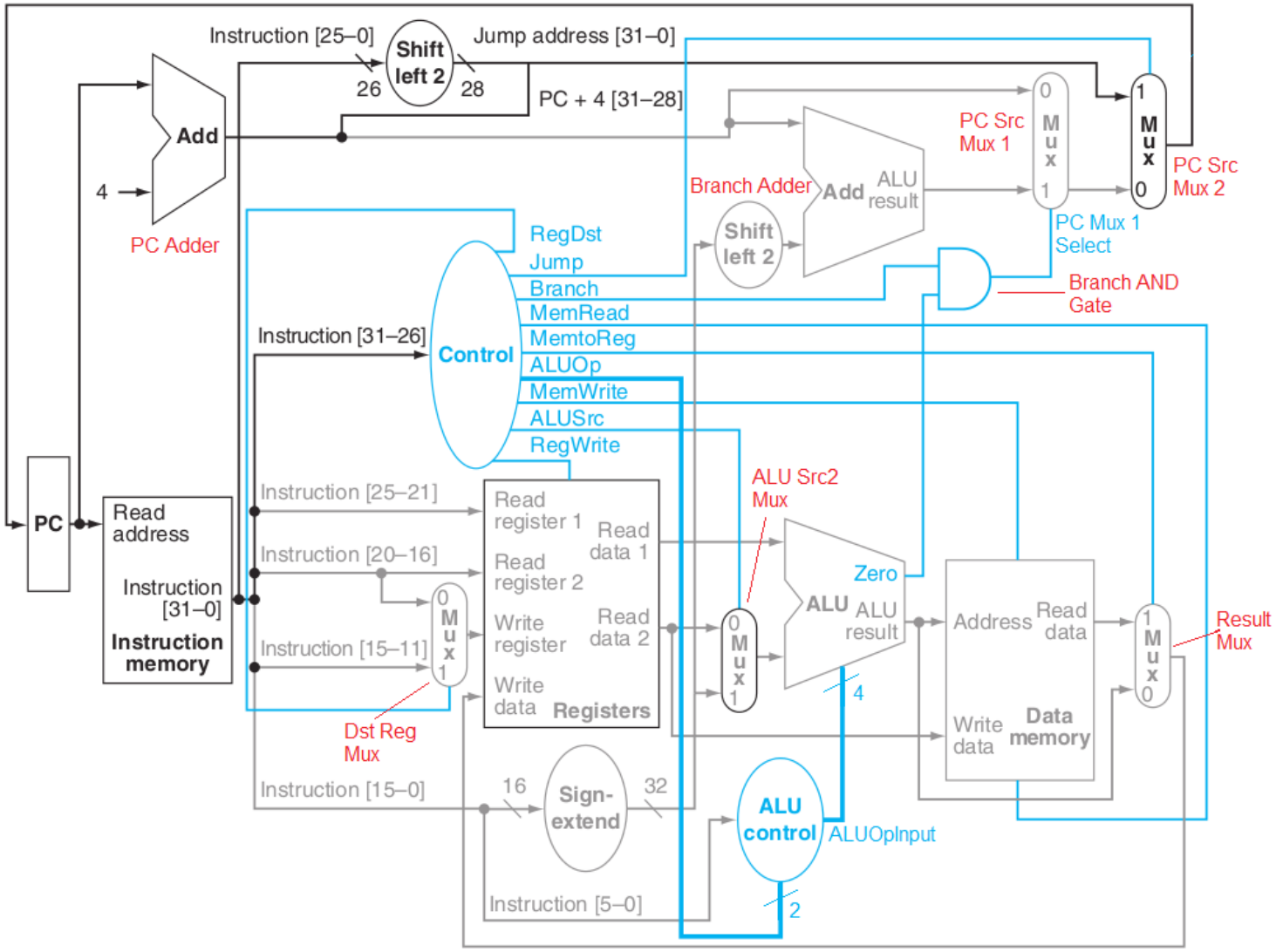
Consequently,

$addr_{25:0} = (0\text{x}0040\_401\text{C} - [0\text{x}0 \ || \ 0\text{x}000\_0000]) >> 2 = (0\text{x}0040\_401\text{C} - 0\text{x}0000\_0000) >> 2$
$addr_{25:0} = 0\text{x}0040\_401\text{C} >> 2$
$addr_{25:0} = 0000\ 0000\ 0100\ 0000\ 0100\ 0000\ 0001\ 1100 >> 2$
$addr_{25:0} = 00\ 0001\ 0000\ 0001\ 0000\ 0000\ 0111$ (discarding the four msb's after shifting)

The `j` opcode is 000010, so the instruction encoding will be:

```
000010 00000100000001000000000111 = 0x08101007
```

Here is the completed single-cycle design block diagram showing the logic in the upper right corner to implement `j`.



4.3.14 Single-Cycle Summary

Every instruction finishes within **one clock cycle**.

This restriction limits each combinational logic block to **one operation** per clock cycle, i.e., at the beginning of the clock cycle, it starts computing some output and must be finished by the end of the clock cycle.

Because each combinational logic block can only be used once, we must **duplicate** some of them. Which ones?

We have three adders (PC Adder, Branch Adder, ALU Adder)
We have two shift-left-by-2 units.

Memory cannot be read and written in the same clock cycle, so **separate** instruction and data memories were required.

The **clock period** and hence **clock frequency** are determined by the time it takes for the bits of the **slowest instruction** (`lw`) to travel through the datapath. The path the bits take is known as the **critical path**.