CS 250: Computer Architecture

Final Exam

Spring 2025

Benjamin Lobos Lertpunyaroj

May 8th, 10:30AM – 12:30PM

Exam contents and details for referencing

- Final exam is held in Fowler Hall on May 8th (Thursday), from 10:30 AM to 12:30 PM.
- Previous cumulative book chapters
  - Chapter 1 sections 1, 2, and 3.
  - Chapter 2, sections 1, 2, 3, 4, 5, 6, and 7.
  - Chapter 3, sections 1, 2, and 5.
  - Chapter 4, sections 1, 2, 3, 4, 5, 6, 7, and 8.
  - Chapter 5, sections 1, 2, 3, 4, 7, and 8.
  - Chapter 8 (Appendix A), sections 1, 2, 3 (but not PLAs or ROMs), 5, 7 (lightly), and 8.
- All lecture notes and lecture slides.
- All labs (1 - 11).
- The order of appearance of contents in this document is arbitrary.

Table of Content

Appendix A

Logic & Gates

An *asserted* signal is logically true, the *deasserted* is the opposite.

Two types of logic systems

Combinational logic: No memory in components, hence same output given same input.

Sequential logic: Memory in components, hence output depends on input and current memory state.

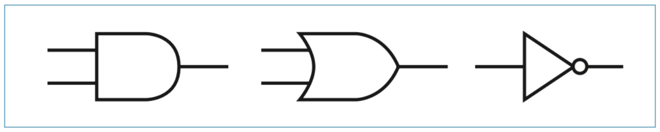


Figure 1: `AND` *gate,* `OR` *gate, and inverter*

The gates can be combined to form different forms of logic. An example of this is $\overline{\overline{A} + B}$ which is equivalent to $A \cdot \overline{B}$ by De Morgan's law, seen in Figure 2.

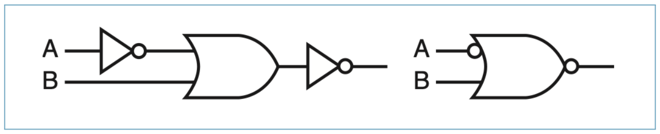
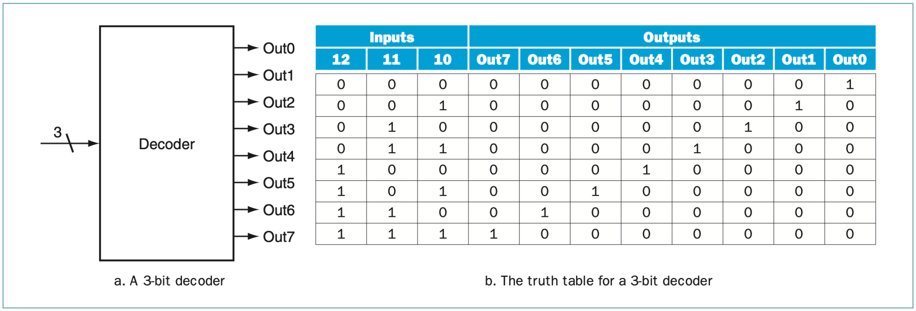


Figure 2: *Logic gate implementation of example formula*

Decoders & Multiplexors

A **decoder** is a logic block that has an n-bit input and 2^n outputs, where there is one unique true bit as output from a unique set of bytes of input.



Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Figure 3: *3-bit input decoder that generates* $2^3 = 8$ *different outputs (Out0 – Out7)*

$$2^n \text{ outputs } \therefore \log_2(\text{output}) = \text{input bits}$$

Encoders are the other way around.

Multiplexors have a selector input (or control value), that will determine which inputs will become outputs.

In the case of the two-input MUX, its representation is the following, $C = (A \cdot \overline{S}) + (B \cdot S)$, using n (data inputs) AND gates, and one OR gate.

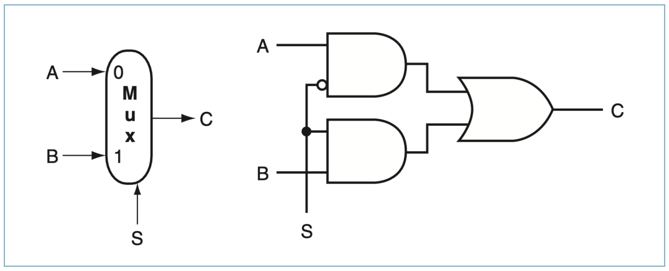


Figure 4: *Two-input multiplexor that generates one output depending on the selector input S*

$$n \text{ (data inputs)} \therefore \log_2 n = S \text{ selector bits required to represent all inputs}$$

Often times a decoder generates n bits for a MUX, to be used as a selector signal.

Buses

A collection of data lines that is treated as a single logical signal.

When showing a logic unit whose inputs and outputs are buses, the unit must be replicated a sufficient number of times to accommodate the width of the input.

You can use multiplexors to select between two buses, requiring n inputs to represent n-bit buses.

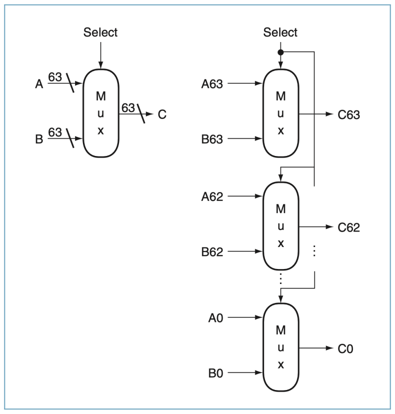


Figure 5: *1-bit multiplexors replicated 64 times to represent two 64-bit buses*

ALUs

Operation done by the ALU

Logic operations: AND and OR gate operations, with NOR being available through an inversion of both input signals with AInvert and BInvert control signals.

Arithmetic operations: Addition and subtraction through the full adder, and BInvert control signal on one input for determining the type of operation.

The LEGv8 word is 64 bits wide, as such a 64 bit wide ALU is required (64 1-bit ALUs).
In its simplest form, a 1-bit logical unit for AND and OR operations simply requires a multiplexor an a one bit control signal to select between the two operations ($2^1 = 2$).

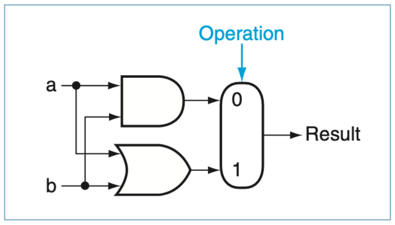


Figure 6: *1-bit logical unit for* AND *and* OR *operations*

Implementing addition requires two input operands, one output, a CarryIn bit carried from the less-significant bits of the operation (i.e. another 1-bit logical unit), and a CarryOut bit to be carried forward to the next more significant bit.

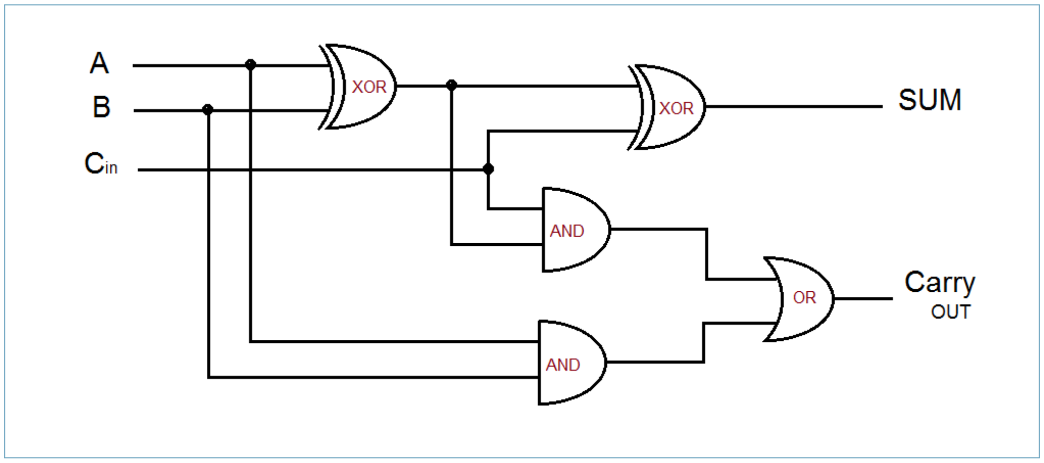


Figure 7: *Full adder that performs mod 2 addition*

The combination of the adder and the logic gates, coupled with a multiplexor with a control signal to determine the operation makes a complete 1-bit ALU, which can be seen in Figure 8.

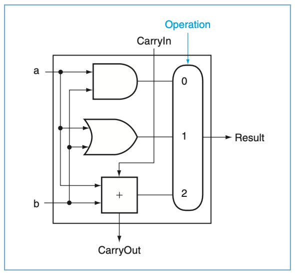


Figure 8: *1-bit alu with logical operations and addition*

For expanding to a 64-bit ALU, the adders have to set up a ripple carry from the least to the most significant bit.

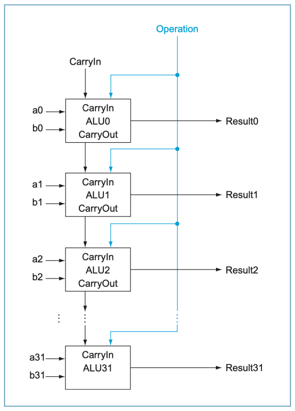


Figure 9: *Ripple carry implemented for a 64-bit ALU*

By inverting the second input (`BInvert` $= 1$, seen in Figure 10) and setting `CarryIn` to 1 in the least significant bit of the ALU, we get two's complement subtraction of `b` from `a`.
To implement a `NOR` function, existing components can be combined, $(\overline{a+b}) = \overline{a} \cdot \overline{b}$ (DeMorgan's theorem), which means we need an `AND` and two inverters for both `a` and `b`, seen in Figure 10

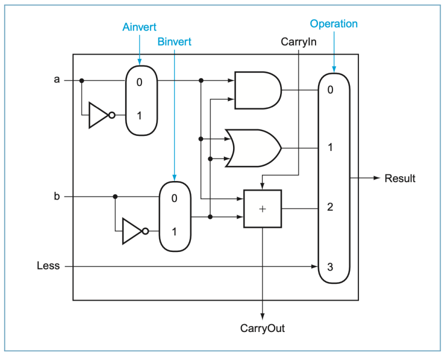


Figure 10: *1-bit ALU that performs subtraction, and* `NOR` *operations*

On a 64-bit ALU we can use a zero flag to help with conditional branch instructions in LEGv8 (e.g.

CBZ), as they receive to inputs and require to test if the subtraction has a zero.

The following represents this with an inversion of an OR tree on all results from the subtraction considering a 64-bit subtraction, fully represented in Figure 11.

$$Zero = \overline{(R_0 + R_1 + R_2 + ... + R_{63})}$$

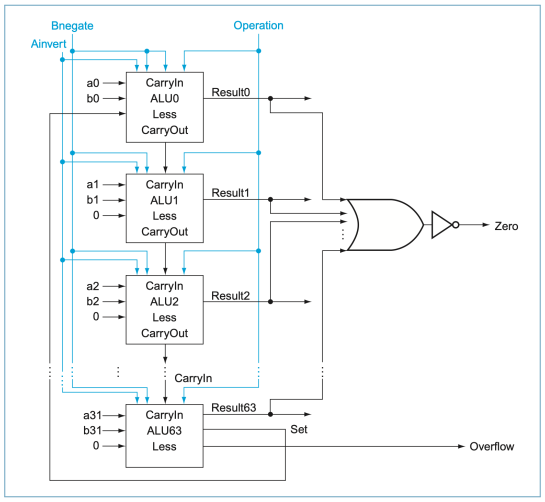


Figure 11: *64-bit ALU* OR *tree and an inverter for determining the Zero flag*

For a generalized symbol of the ALU, Figure 12, where ALU operation is the control signal of the MUX that determines the type of operation.

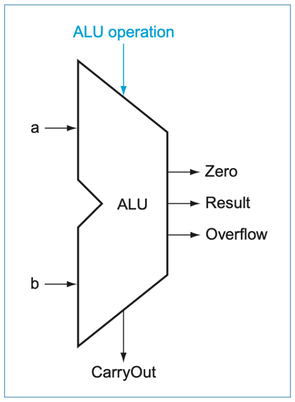


Figure 12: *General symbol for an ALU or an adder*

Clocks

Clocking methodology semantics

Edge triggered clocking: State changes occur on a clock edge.

Synchronous system: Type of memory system where data is read only when a clock signal indicates stability (i.e. non-changing value).

A combinational logic block, recieves an input and then generates an output for a state element which is updated on a clock edge.

An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race condition.

For this to work, the clock cycle must be long enough for the state element to have received a stable input before the next active clock edge.

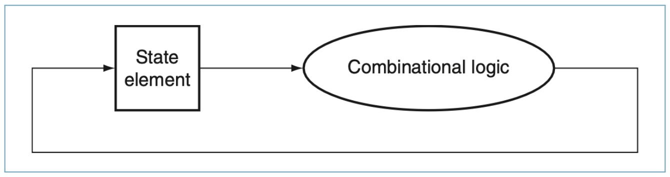


Figure 13: *Edge-triggered state element to be read and written to in one active clock edge*

One such state element is the register file.

Flip-flops & Latches

Types of clocked memory elements

Flip-flops: Edge-triggered element that changes the stored state only at a clock edge.

Latches: Level-sensitive element that changes the stored state at any time the clock is asserted.

Flip-flops are build upon latches and are going to be used in edge-triggered systems.

A **D flip-flop** or **D latch** is used for storing the value of one data input signal, in the internal memory, at the clock edge.

To implement a **D latch**, it requires two inputs, the data to be stored `D`, and the clock signal `C`, producing two outputs, the value of the internal state `Q`, and its complement $\overline{\texttt{Q}}$.

The implementation has cross-coupled `NOR` gates that store the state value unless `C` is asserted, in which case `D` replaces the value of `Q` and is stored.

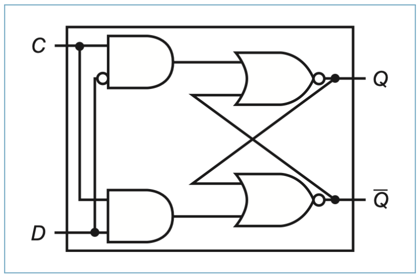


Figure 14: *D latch, composed of crossed* `NOR` *gates and a* `SR` *latch*

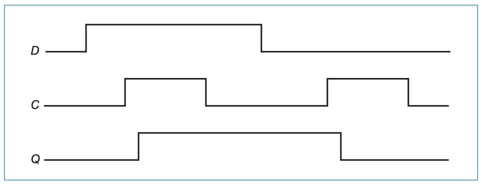


Figure 15: *Progression of a D latch, assuming output is initially deasserted*

To implement a **D flip-flop**, with a falling-edge trigger, we can use two D latches, master and slave. Master sets input `D` when `C` is asserted. When `C` falls, master is closed, but slave is open and gets its input from master's `Q`.

In this sense, the rising-edge represents when the master takes in the `D` value, and the falling-edge represents when the slave takes in the master's `D` producing the final `Q`.

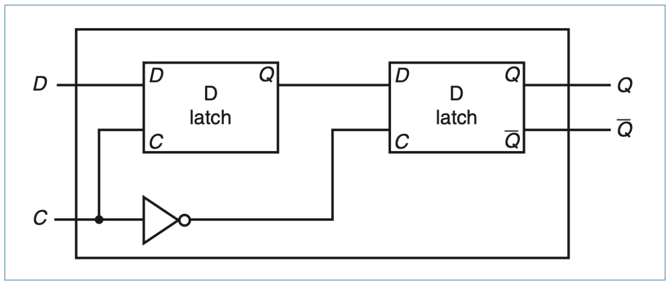


Figure 16: *D flip-flop with a falling-edge trigger made from two D latches, master and slave*

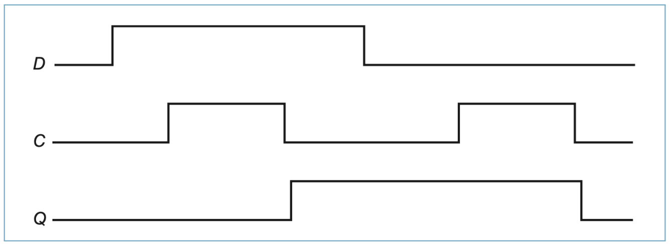


Figure 17: *Progression of a D flip-flop with a falling-edge trigger, where output is initially deasserted*

The minimum time `D` must retain a valid input is the setup time plus the hold time (after edge).

Register files

A **register file** consists of a bunch of **registers** that can be read and written to, and a `WriteReg` control signal (clock).

For writing it requires the control signal, the number of the register to write to (`Write register`), and the data to write (`Write data`).

For reading it requires the numbers of the registers to read from (`Read register number 1 & 2`), and it outputs the read contents from two registers (`Read data 1 & 2`).

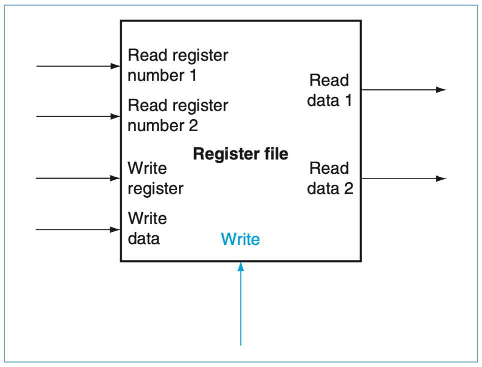


Figure 18: *Register file with two read ports and one write port*

The implementation for the write port consists of a decoder that will select one of the `n - 1` registers that will be `AND`ed with the `WriteReg` signal to act as the `C` input for the registers (D flip-flops). The `D` input for every register is the `Write data` input from the reg. file.

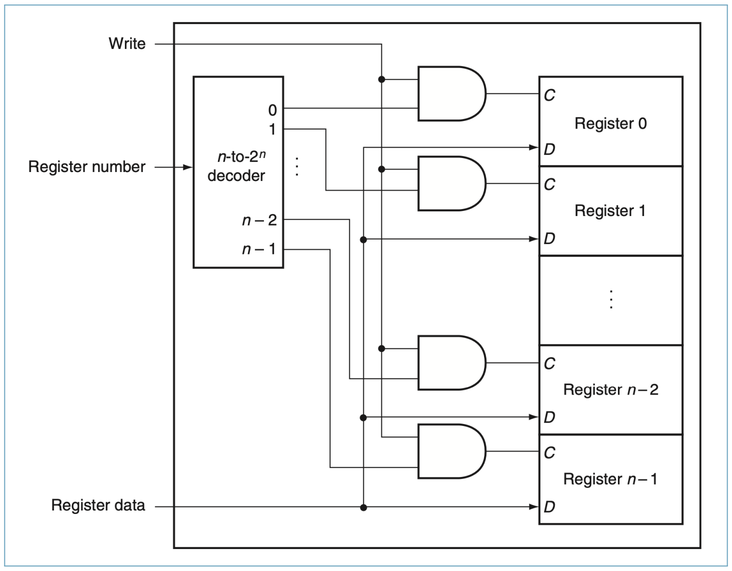


Figure 19: *Write implementation in the register file*

The implementation for the two read ports consists of using the stored state of the registers (`Q` output), as inputs for two different MUXes that use the `Read register number 1 & 2` reg. file inputs as control signals to output the information of the two registers specified.

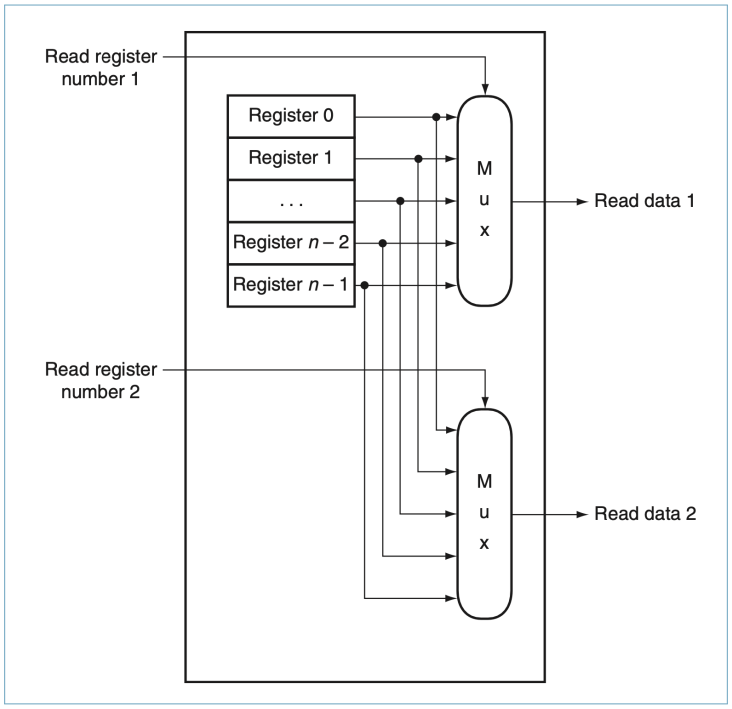


Figure 20: *Read implementation in the register file*

Chapter 2

The size of a register in LEGv8 is 64 bits, which are denominated as doublewords (8 bytes).
A word, the natural unit of access in a computer, is 32 bits (4 bytes).

LEGv8 Assembly

Relevant LEGv8 instructions

Addition:	`ADD X1, X2, X3`
Subtraction:	`SUB X1, X2, X3`
Add immediate:	`ADDI X1, X2, #20`
Subtract immediate:	`SUBI X1, X2, #20`
Load register:	`LDUR X1, [X2, #20]   // Load mem. at addrs. X2 + 20`
Store register:	`STUR X1, [X2, #20]   // Store at addrs. X2 + 20`
Logical AND:	`AND X1, X2, X3`
Logical Inclusive OR:	`ORR X1, X2, X3`
Logical Exclusive OR:	`EOR X1, X2, X3`
Bitwise Left:	`LSL X1, X2, #20`
Bitwise Right:	`LSR X1, X2, #20`
Conditional branch is 0:	`CBZ X1, L0   // Inmediate in operand is in word bytes`
Conditional branch not 0:	`CBNZ X1, L0   // Inmediate in operand is in word bytes`
Branch:	`B L0   // Inmediate in operand is in word bytes`

The above list excludes instructions that set flags, e.g. `ADDS`, `SUBS`, `SUBIS`, `ADDIS`, and zero independent conditional branching, e.g. `B.cond`.

LEGv8 architecture design

There are 32 64-bit registers, limited as larger number of registers implies increasing clock cycle time as electronic signals take longer as they must travel further. It also makes instruction formats more constrained.
Many architectures have alignment restriction that establish that words must start at addresses that are multiples of 4 and doublewords at multiples of 8.

Signed and unsigned numbers

Denomination of bits in a bit string

Least significant bit: The rightmost bit in a bit string. In a doubleword this is bit 63.

Most significant bit: The leftmost bit in a bit string. In a doubleword this is bit 0.

A **sign and magnitude** representation of a signed number has one bit set aside to represent the sign of the integer. Issues come with having a `-0` and `+0`, and more steps needed to determine the sign by the adder.

For making the hardware simple, **two's complement** representation is used, which used leading `0`s to denote positives, and leading `1`s to denote negatives. This means that the most significant bit can be used as a sign bit, making conversions the following way.

$$(x_{63} \cdot -2^{63}) + (x_{62} \cdot 2^{62}) + (x_{61} \cdot 2^{61}) + ... + (x_1 \cdot 2^1) + (x_0 \cdot 2^0)$$

As an example,

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_{\text{two}} = -4_{\text{ten}}$$

Overflow occurs when the sign bit gets overridden, i.e. `0` when negative, `1` when positive.

Using two's complement, sign extension, used for conditional branching, just requires to copy the sign bit `m - n` bits over, where `m` is the new size of the bit string.

Power of 2 prefix definitions

Kibibyte (Kib): 2^{10} bytes.

Mebibyte (Mib): 2^{20} bytes.

Gibibyte (Gib): 2^{30} bytes.

Tebibyte (Tib): 2^{40} bytes.

Pebibyte (Pib): 2^{50} bytes.

Instruction format fields

Opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

Table 1: *R-type format,* `ADD, SUB, LSL, LSR`

Opcode	Address	Op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

Table 2: *D-type format,* `LDUR, STUR`

Opcode	Immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

Table 3: *I-type format,* `ADDI, SUBI`

Opcode	Address	Rd
8 bits	19 bits	5 bits

Table 4: *Conditional branch format,* `CB(N)Z`

Opcode	Address
6 bits	26 bits

Table 5: *Unconditional branch format,* `B`

All instruction field formats are 32 bits, thus the name *32-bit instructions.*

The `opcode` denotes the operation and format of an instruction.

The register operands, `Rn`, `Rm`, and `Rd` are 5 bits to represent the 32 registers ($\log_2 32 = 5$).

The `address` field can represent a region of $\pm 2^8$ bytes around the base register `Rn`.

The `shamt` field represents the shift amount used by bit-shift instructions.

Conditional Statement & Loops

Simple conditional statement implementation

```
    CBNZ X3, Else        // if (X3 == 0) {
    ADD  X0, X1, X2      //     X0 = X1 + X2;
    B    Exit            // }
Else:                    // else {
    SUB  X0, X1, X2      //     X0 = X1 - X2;
Exit:                    // }
```

Simple while loop implementation

```
    ADDI X0, XZR, #10    // X0 = 10;
Loop:
    CBZ  X0, Exit        // while (X0 != 0) {
    SUBI X0, X0, #1      //     X0 -= 1;
    ADD  X1, X2, X0      //     X1 = X2 + X0;
    B    Loop            // }
Exit:
```

For a more complete example, the following converts the C code into LEGv8 assembly instructions.

C code loop into LEGv8 ASM

```
// C
int i = 0;
int k = 10;
while (save[i] == k) {
    i += 1;
}

// X25: Address of save[]

// Textbook LEGv8 ASM
    ADDI X22, XZR, #0   // EOR X22, X22, X22
    ADDI X14, XZR, #10
Loop:
    LSL  X10, X22, #2
    ADD  X10, X10, X25
    LDUR X11, [X10, #0]
    SUB  X12, X10, X14
    CBNZ X12, Exit
    ADDI X22, X22, #1
    B    Loop
Exit:

// Alternate LEGv8 ASM
    ADD  X10, XZR, X25
    ADDI X11, XZR, #10
Loop:
    LDUR X12, [X10, #0]
    SUB  X12, X12, X11
    CBNZ X12, Exit
    ADDI X10, X10, #8
    B Loop
Exit:
```

You can "throw away" the result by writing into `XZR`, the zero register.

Set Flag Branching

The conditional branch instruction, `B.cond`, allows for `.cond` to be used for signed comparisons, `EQ` (equals), `NE` (not equal), `LT` (less than), `LE` (less than or equal), `GT`, or `GE`.

Also it allows for unsigned comparisons, `LO`, `LS` (lower or same), `HI`, or `HS` (higher or same).

The flag for comparison is set by instructions like, `ADDS`, `ADDIS`, or `SUBS`, but they are limited in number as they create dependencies that obstruct pipelining execution.

A use case is bounds checking, that is, $0 \leq x < y$.

Bounds checking shortcut in LEGv8 ASM

```
// Pseudo C
if (X20 >= X11 || X20 < 0) {
    goto Error;
}

// LEGv8 ASM
    SUBS XZR, X20, X11
    B.HS Error
    B    Exit
Error:
    // Error handling
Exit:
```

Signed negative numbers look massive when looked at from an unsigned comparison in two's complement, and it also allows for checking if it is below a number (`X11` in example).

Chapter 3

Floating point representation

Compromise must be found between the size of the *fraction* and the *exponent*.

Tradeoff of floating-point representation

Precision: Increased by an increase in the size for the *mantissa* or *fraction*.

Range: Increased by an increase in the size for the *exponent*.

A LEGv8 implementation of **floating-point** numbers has `s` as the sign bit, `exponent` an 8-bit with bias, and `fraction` a 23-bit number.

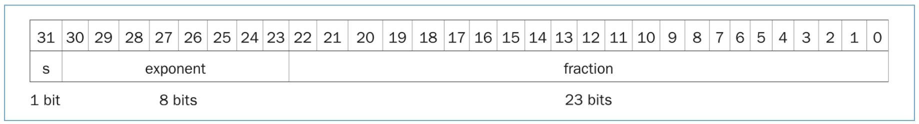


Figure 21: *LEGv8 floating-point representation*

$$(-1)^S \times F \times 2^E$$

Range of representation is $2_{\text{ten}} \times 10^{38}$ considering 2^7(exponent 8-bit rep. divided by 2 for bias) ≈ 38
Overflow entails the exponent being too large for the `exponent (E)` field to represent it. Underflow is the same situation but the `exponent` field representing a negative value.
To reduce the chances of underflow or overflow we use **double precision**, which is a floating-point value represented in a 64-bit doubleword.

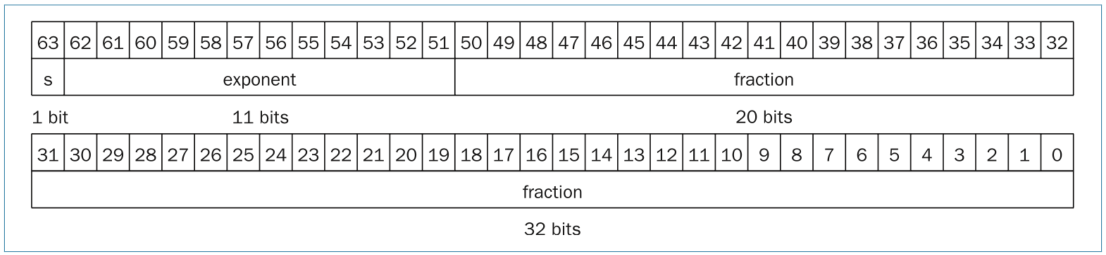


Figure 22: *LEGv8 double precision floating-point representation*

In the case of an overflow/underflow an interrupt (unscheduled disruption call) saves the address of the instruction (to resume after correction) and jumps to a predefined address that deals with the exception.

IEEE 754 Floating-Point Standard Specifications

The floating-point can always have a leading `1.0`, as such IEEE assumes it as a hidden bit increasing the representation for the `fraction` by 1 (23 to 24 in single p., or 52 to 53 in double p.).

$$(-1)^S \times (1 + F) \times 2^{(\text{Exponent}-\text{Bias})}$$

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + ...) \times 2^E$$

IEEE 754 has `NaN` for invalid operations (e.g. $\frac{0}{0}$). Infinities can be represented through the largest exponent representations $(+/-)$ instead of interrupting.

The sign bit is the most significant bit to easily process integer comparisons.

For representing the exponent without needing a sign bit, IEEE 754 uses a bias of 127 for single, and 1023 for double, that subtracts from the total possible representation to denote negative and positive exponents.

So for single precision, an exponent of -1 is $-1 + 127_{\text{ten}} = 126_{\text{ten}}$, and $+1$ is $1 + 127_{\text{ten}} = 128_{\text{ten}}$

Floating-Point Addition

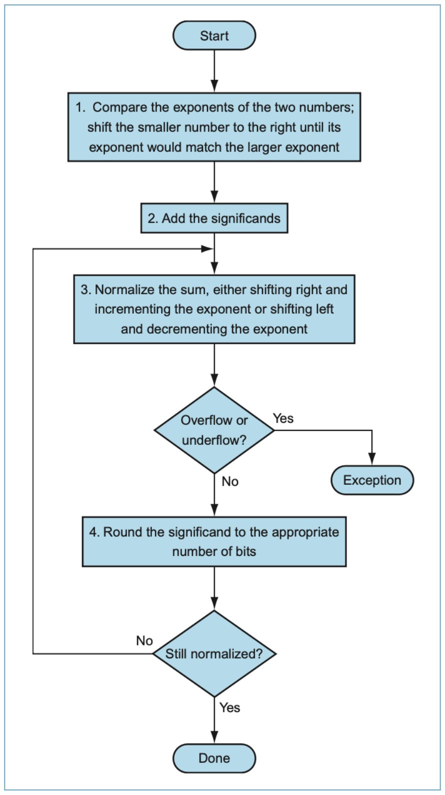

Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Figure 23: *Floating-point addition logic path*

Chapter 1

Design principles

→ **Design with Moore's Law**

Moore's Law states that an integrated circuit resources double every 18–24 month.
Thus, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts.

→ **Abstraction to Simplify Design**

Abstractions to hide lower-level details to simplify higher-level representations (more productive).

→ **Common case fast**

If you know what the common case is, enhancing it over the rarer cases is more optimal, as it involves more and is usually easier to enhance.

→ **Performance via Parallelism**

Computing operations in parallel (at the same time) increases performance over a time period.

→ **Performance via Pipelining**

Pipelining is a pattern of parallelism that involves increasing the throughput of a certain process (i.e. moving up the chain faster).

→ **Performance via Prediction**

If you can predict successfully to a degree you can anticipate instead of wait. In some cases it can be more beneficial to guess than to stall when misprediction costs are reasonable.

→ **Memory hierarchy**

The most expensive and fast memory per bit is at the top, and the opposite is at the bottom. Caches (top) can give the illusion of fast main memory (bottom).

→ **Dependability via Redundancy**

Systems become dependable when there is redundancy to account for solving failures and failure detections.

Amdahl's Law

Related to the design principle of *making the common case fast*, you only benefit from improving the part you actually use, which is illustrated in **Amdahl's law**.

Amdahl's Law

Law: Performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

$$S_{\text{total}} = \frac{1}{(1-f)+\frac{f}{S_e}}$$

Splitting up the original time into the unimprovable part $1-f$ and the improvable part f, speeding up the f portion by S_E yields a time ratio of

$$\frac{T_{\text{new}}}{T_{\text{old}}} \to (1-f)+\frac{f}{S_E}$$

As $S_E \to \infty$, the maximum overall speedup is $\frac{1}{1-f}$.
Thus, taken as the ratio between the old and new speeds, the formula for Amdahl's Law emerges.

Chapter 4: Datapath

Logic Design Elements

Combinational elements: Do not store any information internally, such as an ALU, and logic gates.

State elements: Have internal memory, a *state*, that remembers even in power loss, such as registers or memory.

Branching in the Datapath

Conditional branching is done by incrementing the PC (program counter, keeps track of the current instruction) by the word (4 bytes) alligned offset (64-bits).

```
B offset     CBZ  Rd, offset
```

The `offset` varies in bits by instruction, requiring a sign-extension (from 32 to 64 bits), and 2 left shifts (word aligned multiple of $4 = 2^2$).

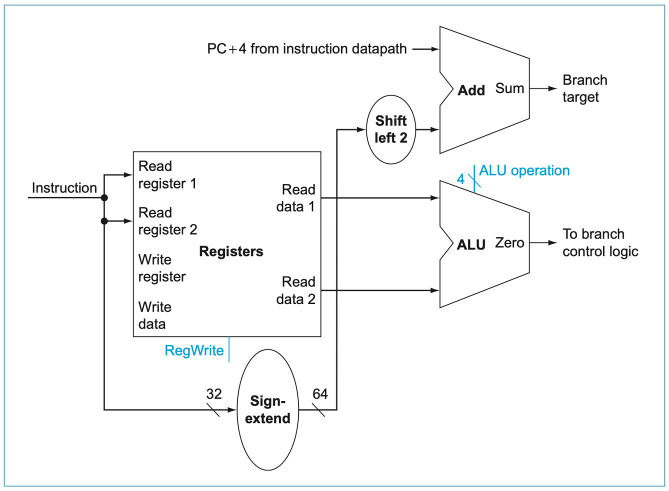


Figure 24: *Subset datapath for unconditional and conditional branching*

Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the zero output of the ALU, seen in Figure 24.

Data memory unit and sign extensions

The sign extend takes the whole 32-bit instruction and selects the specific offset field size (e.g. 9 bits for loads and stores (small offset needed), 19 bits for `CBZ`), and fills upper bits with the most significant bit of the extracted address.

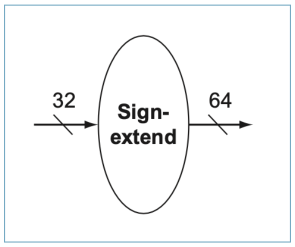


Figure 25: *Sign extension from 32 bits to 64 bits*

For memory instructions (e.g. `LDUR`, `STUR`), we need to write and read from memory using the data memory unit.

The structure takes in the address of memory to access, write input (when storing), read output (when loading), and read and write control signals.

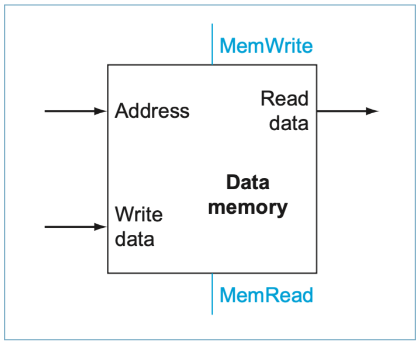


Figure 26: *Memory data unit diagram*

R-type, Branch, and Memory Instruction Datapath

Building the datapath for R-type (ALU operations), conditional branching, and memory instructions in a single clock cycle.

MUX Controlled by the ALUSrc Signal

R-type: Uses the value to do an ALU operation with the first register read output.

Load/Store (D-type): Uses the sign extended `offset` with the first output, that holds the memory address to load/store from. In this case, the second read register output is still relevant in storing, as it holds the data to be written.

MUX Controlled by the MemtoReg Signal

R-type: Skips the reading stage from the data memory unit and sets the output of the ALU to be the `Write Data` input of the register file.

Store (D-type): Reads at `address + offset` from the data memory unit, and redirects the output to the `Write Data` input of the register file.

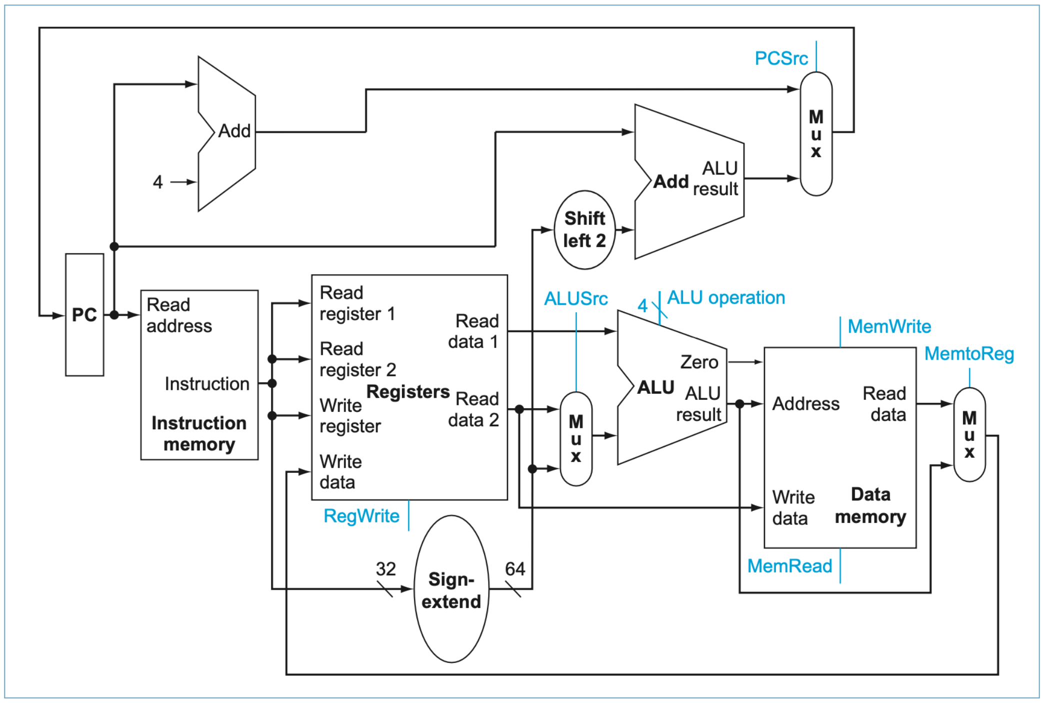
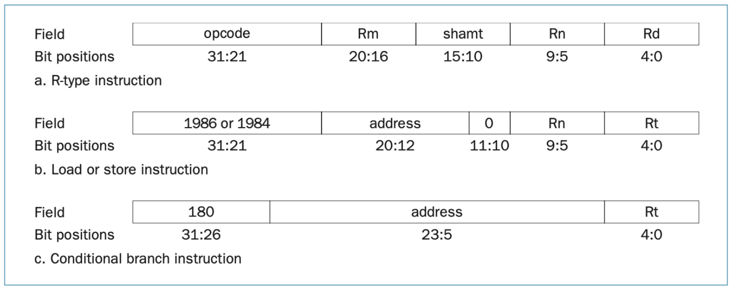


Figure 27: *Datapath for R-type, load/store, and branching instructions*

Instruction Paritions by Instruction Memory

Considering the R-type, load/store, and conditional branching formats, they are broken down as seen in Figure 28.



Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0

a. R-type instruction

Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0

b. Load or store instruction

Field	180	address	Rt
Bit positions	31:26	23:5	4:0

c. Conditional branch instruction

Figure 28: *Memory data unit diagram*

Simplicity favors regularity: The more uniform the design, the simpler the control logic for decoding the instruction (Instruction Memory).

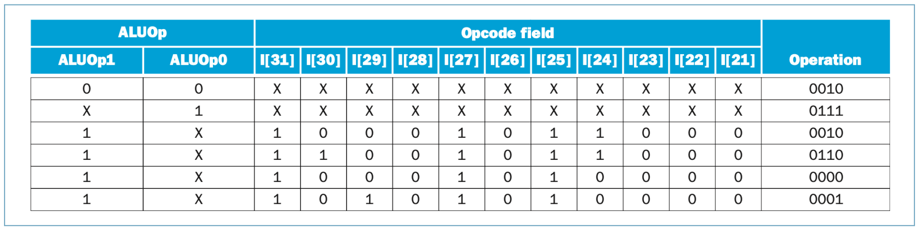
Generating the ALU Control Input

The ALU control is made up from the `opcode` (11-bit) and a 2-bit control signal `ALUOp`.

The `opcode` field only has three bits that matter when determining function (30, 29, 24).

`ALUOp` encodes for addition (`0 0`), passing input B considering `CBZ` (`X 1`), and other functions (`1 X`).

$$\texttt{ALU operation}\ (4) = \texttt{ALUOp}\ (2) + \texttt{opcode}\ (11 \text{ but uses } 3)$$



ALUOp		Opcode field											
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[24]	I[23]	I[22]	I[21]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	X	0111
1	X	1	0	0	0	1	0	1	1	0	0	0	0010
1	X	1	1	0	0	1	0	1	1	0	0	0	0110
1	X	1	0	0	0	1	0	1	0	0	0	0	0000
1	X	1	0	1	0	1	0	1	0	0	0	0	0001

Figure 29: *Truth table for the 4 ALU control bits, with additional "don't care entries"*

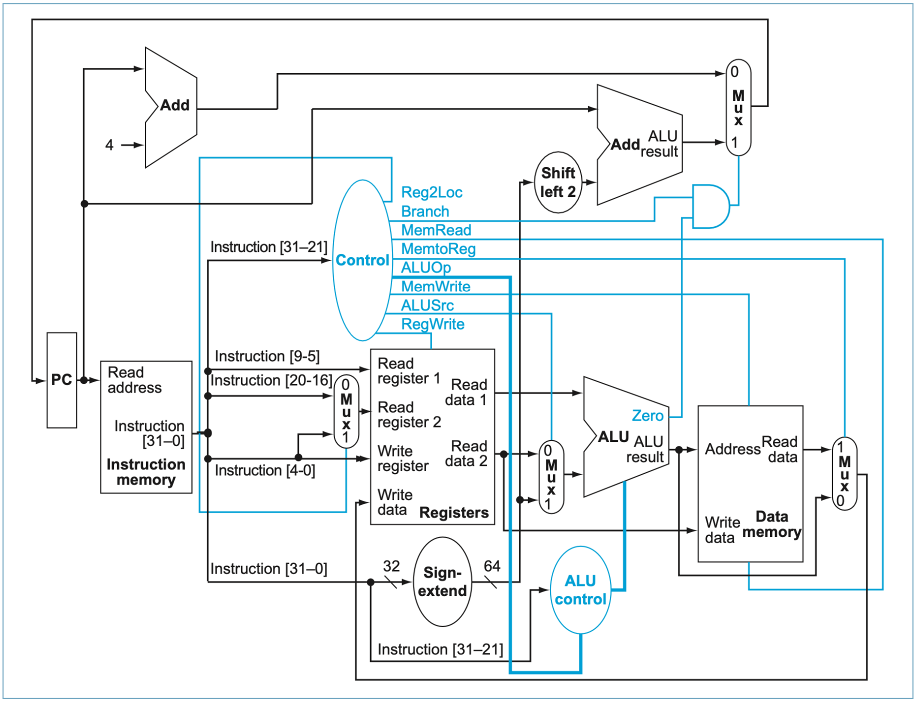
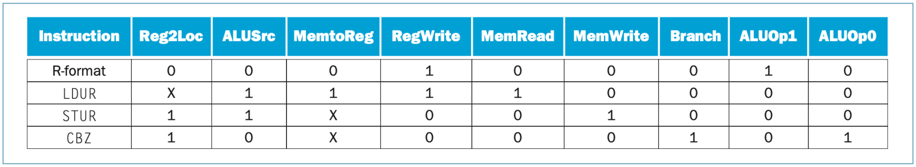


Figure 30: *Datapath for R-type, D-type, and branching instructions with all 9 control lines*

Consider in Figure 30, the `PCSrc` signal, that determines whether to increment PC by 4 or by the offset specified, does not originate from the **control unit**, but rather the `branch` signal from the control unit being asserted and the zero output of the ALU is asserted.



Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

Figure 31: *Setting control lines according to desired instruction (determined by opcode)*

Implementing Unconditional Branching into Datapath

Adding a new signal `UncondBranch` to the control unit and `OR`ing it with the resulting `PCSrc` control signal that emerges from the `branch` control signal.

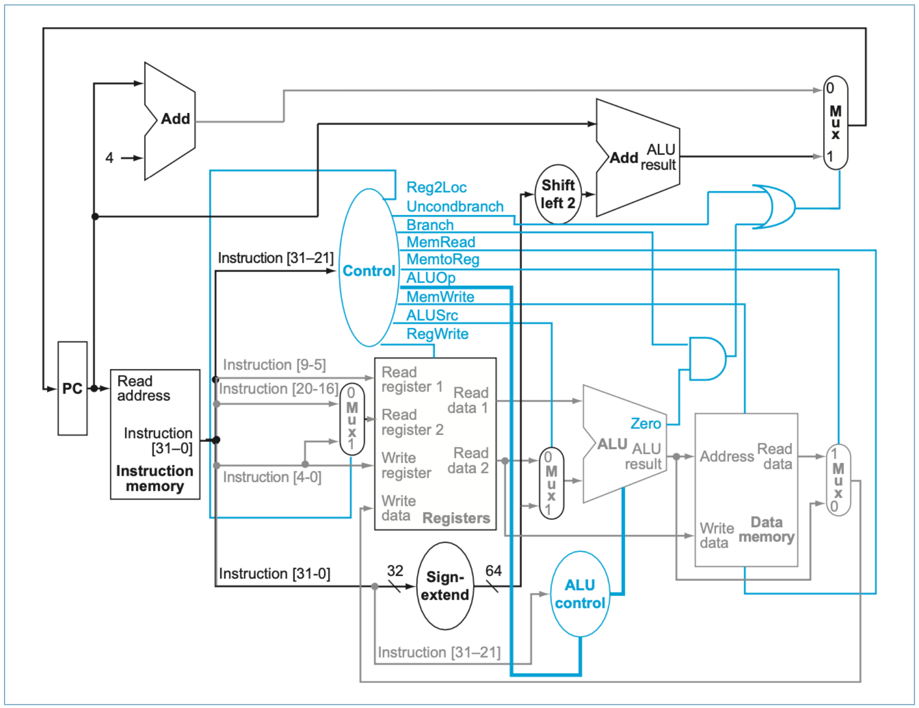


Figure 32: *Datapath with UncondBranch control signal*

Single cycle implementations have the limitation that the clock cycle needs to be greater than the time it takes to complete the longest possible path (load instruction: uses the IM, RegFile, ALU, DataMemory, and the RegFile again).

Chapter 4: Pipelining

Pipelining overlaps instructions in execution.

LEGv8 Pipelining Steps

1. **IF**: Fetch instructions from memory.
2. **ID**: Decode instructions and read registers.
3. **EX**: Executes the op. or calculates the address.
4. **MEM**: Access data memory (if needed).
5. **WB**: Writing back into register (if needed).

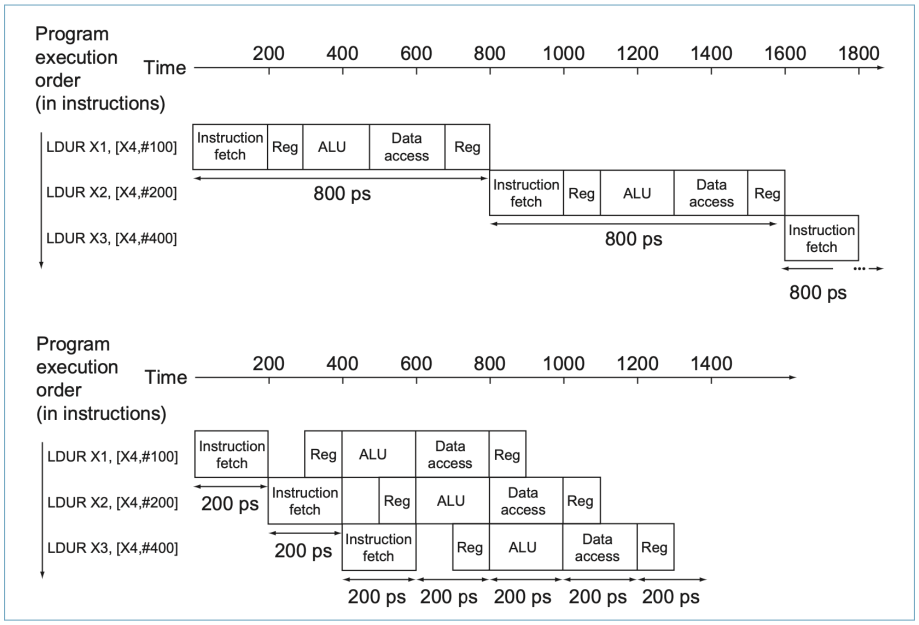


Figure 33: *Single-cycle execution vs Pipelined execution, showing an improvement from 800ps to 200ps*

Pipelining improves performance by increasing instruction *throughput* instead of decreasing execution times by instruction.

Pipelining the Datapath

The separation of the five stages can be seen in the datapath below.

The left to right flow has some exceptions. Write backs involve putting a result in the middle of the path. The selection of the branch that involves +4 or jumping.

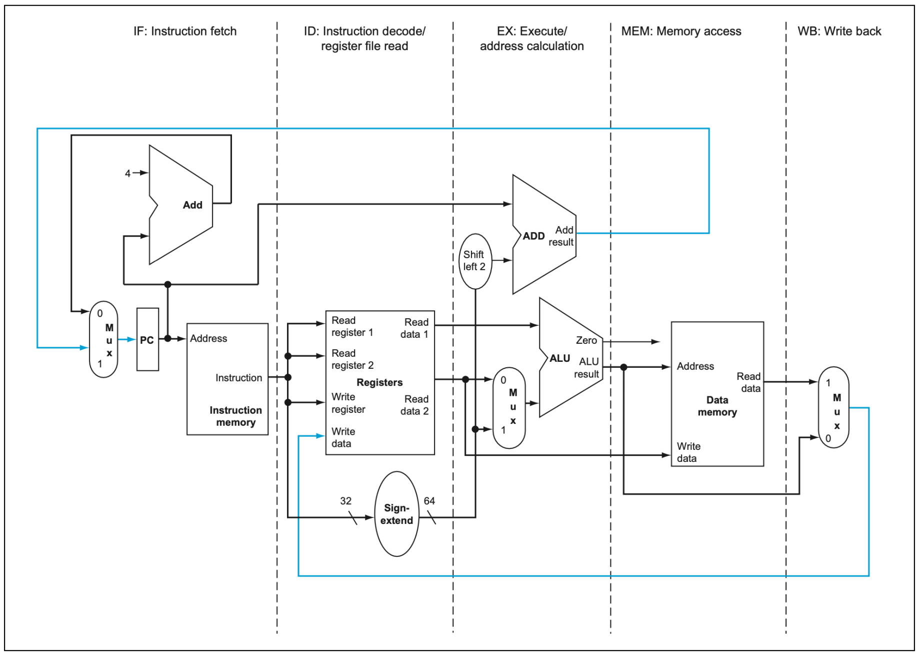


Figure 34: *Five stages separation in Datapath*

Pipeline registers separate the five stages and prevent data loss, keeping stability for newer instructions to use.

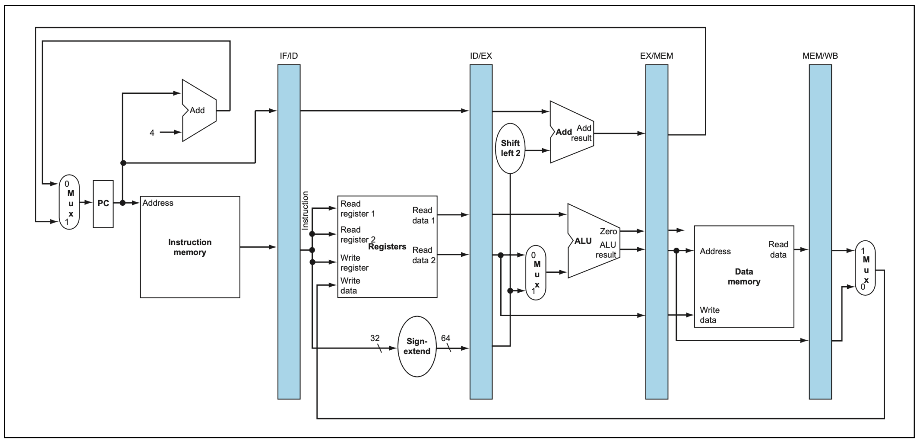


Figure 35: *Pipeline registers in Datapath: IF/ID, ID/EX, EX/MEM, MEM/WB*

Each logical component (Inst. Mem, RegFile, ALU, etc) must be used only once in a single pipeline stage to avoid *structural hazards.*

As instructions, like loads, need to keep track of a write back register, `Rt`, which means that this information needs to be propagated all the way upto the MEM/WB stage.

Propagation also happens with the PC, for calculating branching.

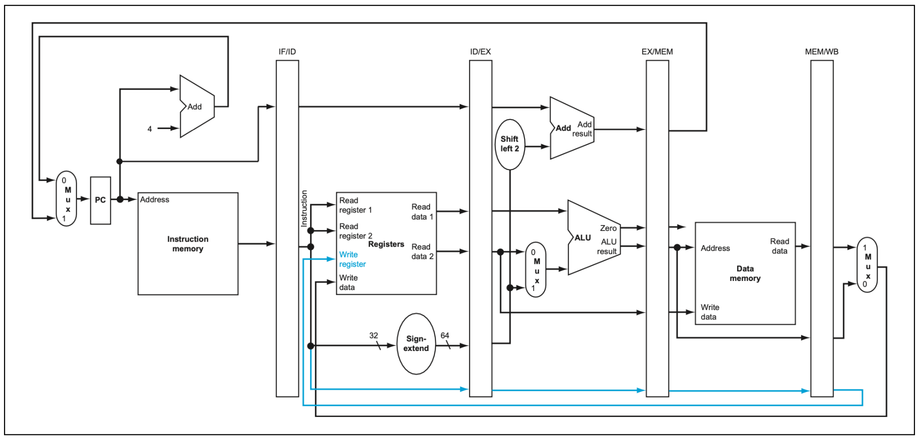


Figure 36: *Pipelined Datapath with propagates write back register*

Pipelining the Control

As each control line is associated with one component active in one pipeline stage, the five stages can be used to classify the existing control signals. This means that pipelined registers can be expanded to fit the control signals.

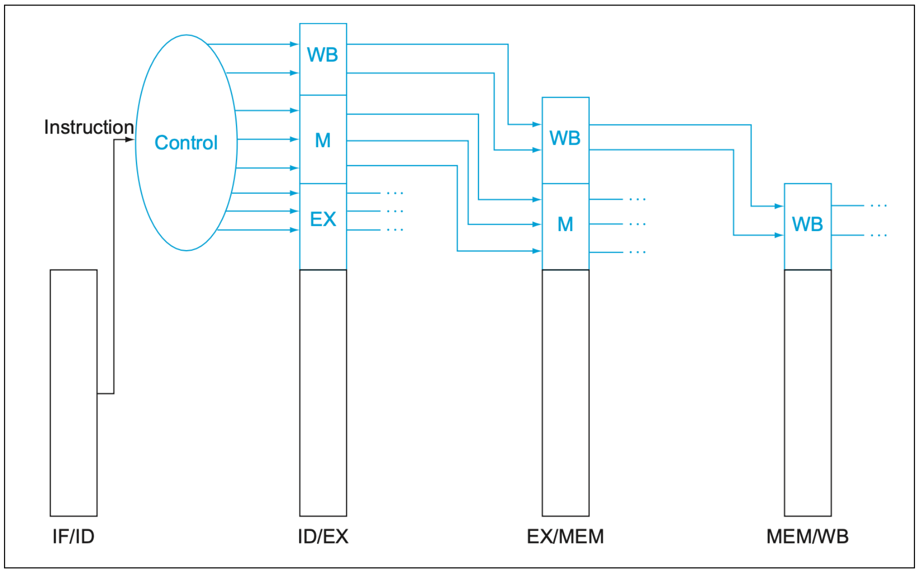


Figure 37: *Control signals with applied Pipelining*

Considering that the control signals start at EX, the control information can be created at ID. The `Reg2Loc` signal is directly encoded into the instruction field as a bit that determined whether to use the `Rm` for R-type or `Rt` for cond. branching and load/store instructions.

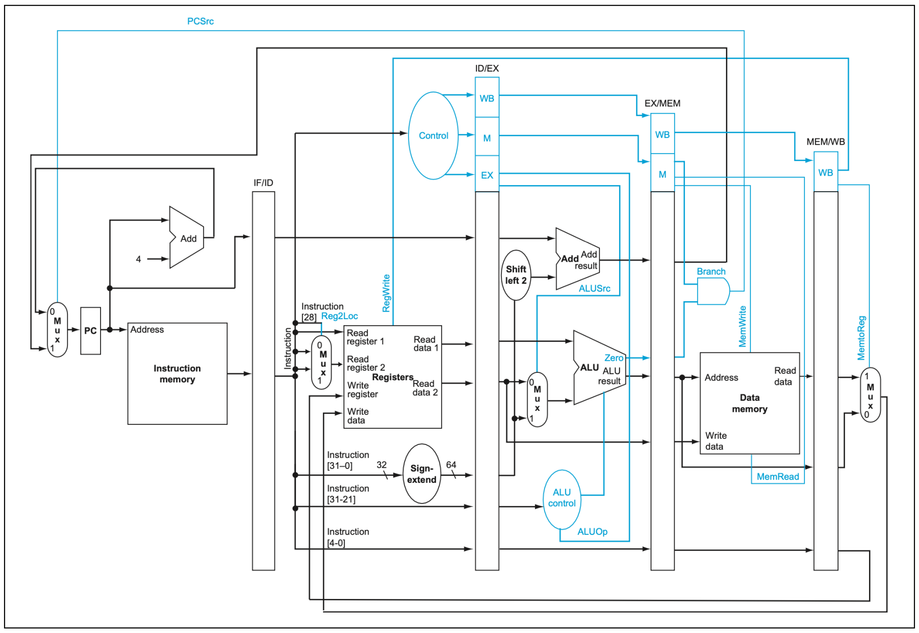


Figure 38: *Pipelined Datapath with Pipelined Control Signals*

Pipeline Hazards

→ **Structural Hazard**

When one element cannot support two instructions at the same time.

Imagined if there was no distinction between IM and Data Memory, which would cause a first instruction to access data, while a fourth one is trying to fetch it at the same time.

→ **Data Hazards**

When a new instruction needs data from another instruction that has not yet finished executing.

Data dependence from one instruction to an earlier one.

```
ADD    X19, X0, X1
SUB    X2, X19, X3    // X19 depends on ADD instruction's execution
```

Stalling would cause a waste of 3 clock cycles, and compilers opts are not enough.

A solution is to **forward** or bypass, which entails, in the example, to supply the result of the ALU directly into the input of the subtraction (or RegFile write).

There are cases where pipeline stalls or bubbles are present in forwarding, such as the **load-use data hazard**, in which the data has not yet been loaded when they are needed by another instruction.

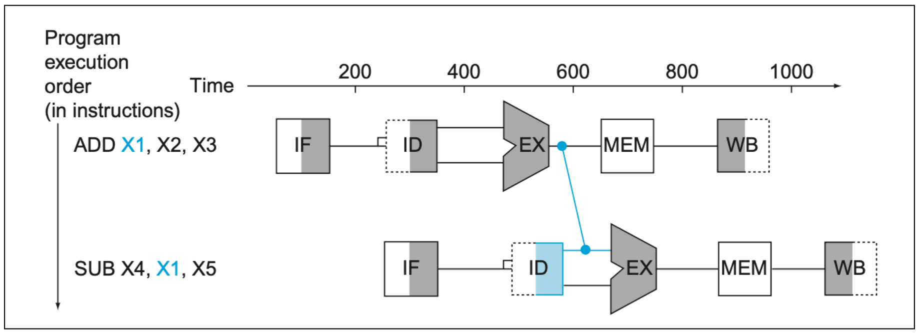


Figure 39: *forwarding implementation in* add *and* sub *example*

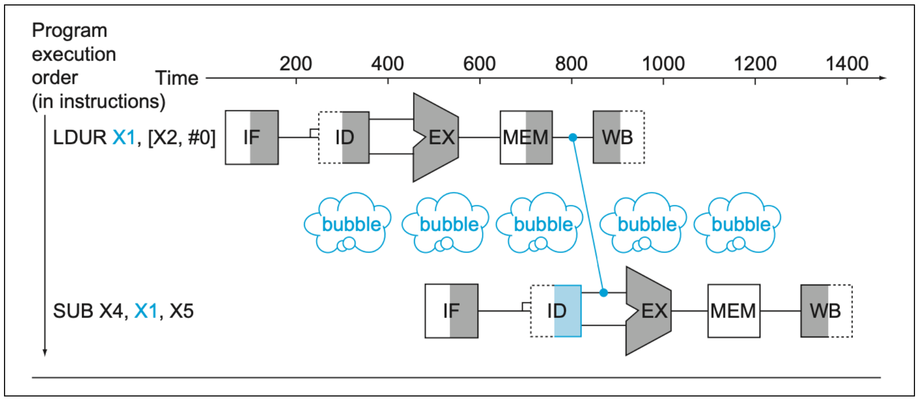


Figure 40: *Load-use hazard with one cycle stalling to forward data*

→ **Control Hazards**

When the fetched instruction is not the one needed (e.g. encountered branching).

Prediction is used to assume rather than wait for the branch to accertain the outcome, such as always not taking the branch.

Dynamic prediction differs in result depending of what happened before (keeping a history).

The penalty for an incorrect prediction is that the pipeline must restart from the proper branch address, stalling the process.