Lecture Notes: ARM64 Architecture and Programming for Raspberry Pi 4B

Ji, Yong-Hyeon

November 8, 2024

Contents

1 Introduction to ARM64 Architecture

ARM64 (AArch64) is a 64-bit architecture developed by ARM Holdings, widely used in modern mobile devices and servers. ARM64 offers a significant register file structure, with 31 general-purpose registers and special-purpose registers such as the Program Counter (PC) and Stack Pointer (SP). These registers provide flexibility for performing low-level operations efficiently.

1.1 Key Features

- 64-bit general-purpose registers (X0-X30).
- Special-purpose registers for the stack, program control, and more.
- Optimized function calling convention.

1.2 Integer Representation in Computer Systems

In computer systems, integers are represented using a fixed number of bits, typically grouped into bytes. This section discusses how integers are encoded, with a focus on binary representation, signed integers, and the implications of overflow.

1.2.1 Binary Representation of Unsigned Integers

An *unsigned integer* of n bits is an integer value that can take on values in the range $[0, 2^n - 1]$. In binary form, such an integer is expressed as a sum of powers of two:

$$I = \sum_{i=0}^{n-1} b_i \cdot 2^i,$$

where $b_i \in \{0, 1\}$ represents the i-th bit of the integer, with b_0 being the least significant bit (LSB) and b_{n-1} the most significant bit (MSB).

For example, a 4-bit unsigned integer can be represented as:

$$I = b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0,$$

which results in possible values in the range $[0, 15]$.

Binary	Decimal	Explanation
0000	0	All bits are zero
0001	1	$1 \cdot 2^0$
0010	2	$1 \cdot 2^1$
0111	7	$1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$
1111	15	Maximum 4-bit unsigned integer

Table 1: Example values for 4-bit unsigned integers.

1.2.2 Signed Integers: Two's Complement Representation

In practice, computers often need to represent signed integers, i.e., integers that can be both positive and negative. One of the most widely used methods for representing signed integers is the *two's complement* notation. In this system, an n-bit signed integer I is expressed as:

$$I = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i.$$

Here, b_{n-1} is the sign bit, with $b_{n-1} = 0$ representing a non-negative integer, and $b_{n-1} = 1$ representing a negative integer.

Binary (4-bit)	Decimal	Explanation
0000	0	Zero, all bits are zero
0111	7	Largest positive number ($2^2 + 2^1 + 2^0$)
1000	-8	Smallest negative number (-2^3)
1101	-3	$-2^3 + 2^2 + 2^0$
1111	-1	All bits are one ($-2^3 + 2^2 + 2^1 + 2^0$)

Table 2: Example values for 4-bit signed integers using two's complement.

1.2.3 Detecting Overflow in Two's Complement Addition

For two signed integers x and y represented in two's complement, overflow in addition occurs if:

$$\begin{aligned} &x > 0 \quad \text{and} \quad y > 0 \quad \text{and} \quad x + y < 0, \quad \text{or} \\ &x < 0 \quad \text{and} \quad y < 0 \quad \text{and} \quad x + y > 0. \end{aligned}$$

This can be efficiently detected by examining the sign bit of the result compared to the sign bits of the operands.

x (Binary)	y (Binary)	Sum (Binary)	Overflow?
0110 (6)	0101 (5)	1011 (-5)	Yes (Positive overflow)
1001 (-7)	1110 (-2)	0111 (7)	Yes (Negative overflow)
0011 (3)	0010 (2)	0101 (5)	No
1101 (-3)	1100 (-4)	1001 (-7)	No

Table 3: Overflow detection in two's complement addition.

1.2.4 Summary

The representation of integers on a computer is constrained by the finite number of bits available. Unsigned integers use all bits to represent non-negative values, while signed integers often employ two's complement to encode both positive and negative values. Careful handling of overflow is essential in arithmetic operations to ensure correct computational results.

1.3 Memory Layout in Computer Systems

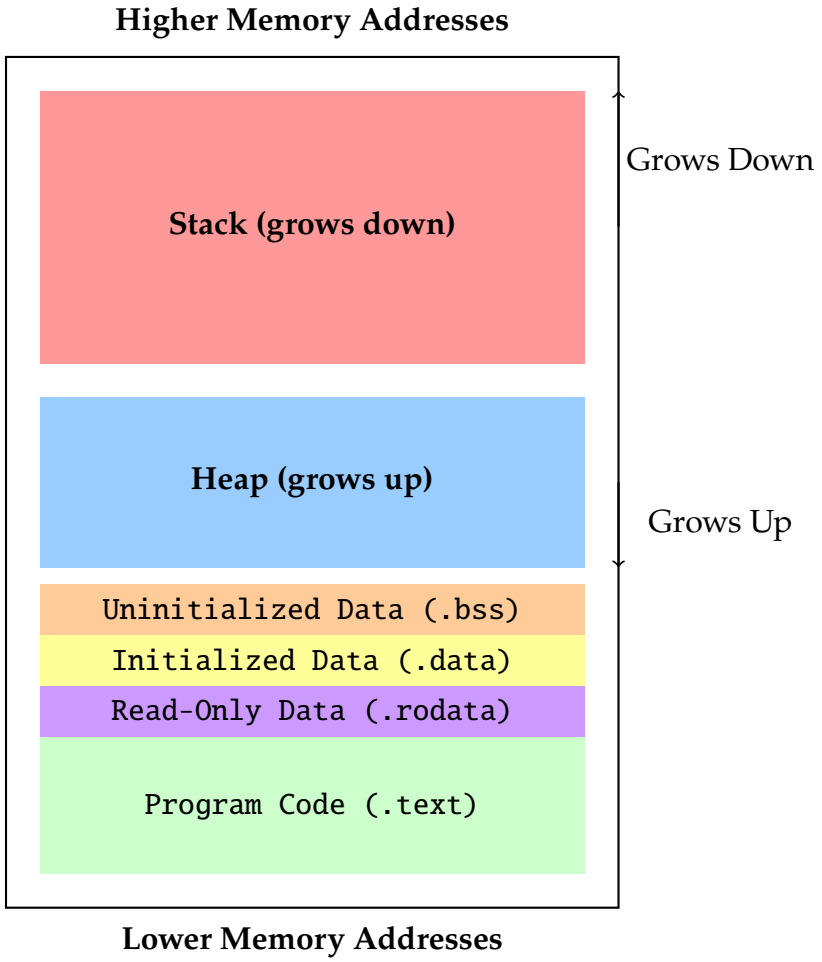
In modern computer systems, memory is divided into several distinct segments that each serve a different purpose in the execution of a program. These segments include the stack, heap, and various regions that hold code and data. This section provides a mathematical description of the layout of memory in a typical system and explains the role of each memory segment.

1.3.1 Memory Segments

A typical program is divided into the following memory segments:

Stack	Used for local variables, function call management, and control flow. It grows downward in memory.
Heap	Used for dynamic memory allocation (e.g., via `malloc`). It grows upward in memory.
.text	Stores the program's executable instructions (machine code).
.data	Stores initialized global and static variables.
.bss	Stores uninitialized global and static variables. This segment is initialized to zero at runtime.
.rodata	Stores read-only data, such as string literals or constant variables.

The memory layout in a typical process can be visualized as:



1.3.2 The Stack Segment

The *stack* is used to store local variables, function parameters, return addresses, and control flow information during program execution. For each function call, a *stack frame* is allocated, which contains the function's local variables and bookkeeping information (such as the return address). The stack grows downward, meaning that memory addresses decrease as more data is added to the stack.

Mathematically, the size of the stack at any given time can be described as:

$$S(t) = S_0 - \sum_{i=1}^{n} f_i,$$

where S_0 is the initial stack pointer, n is the number of function calls made, and f_i represents the size of the i-th function's stack frame.

1.3.3 The Heap Segment

The *heap* is used for dynamic memory allocation, typically requested via system calls such as `malloc` or `new`. Memory in the heap is managed manually by the programmer and persists until it is explicitly deallocated. The heap grows upward in memory.

At any point in time, the size of the heap $H(t)$ can be expressed as:

$$H(t) = H_0 + \sum_{i=1}^{m} d_i - \sum_{j=1}^{k} r_j,$$

where H_0 is the base address of the heap, d_i represents the size of the i-th dynamically allocated block, and r_j represents the size of the j-th deallocated block. The heap size grows with allocations and shrinks with deallocations.

1.3.4 The .text Segment

The `.text` segment contains the program's machine code instructions. It is typically read-only, and any attempt to write to this segment will result in an error. The size of the `.text` segment is constant after the program is loaded into memory.

Let the `.text` segment size be denoted by T. If the program contains n instructions, each requiring b_i bytes of memory, the total size of the `.text` segment is:

$$T = \sum_{i=1}^{n} b_i.$$

1.3.5 The .data Segment

The `.data` segment contains initialized global and static variables. It is writable and persistent throughout the program's execution. Each variable in this segment has an initial value, set by the program's code. The size of the `.data` segment is determined by the number and size of initialized global variables.

If a program has n initialized global or static variables, and each variable v_i requires s_i bytes, the size of the `.data` segment is:

$$D = \sum_{i=1}^{n} s_i.$$

1.3.6 The .bss Segment

The `.bss` segment contains uninitialized global and static variables, which are automatically initialized to zero by the system at runtime. This segment does not require any space in the executable file but consumes memory at runtime.

Let the number of uninitialized variables be n and let each variable v_i require s_i bytes. The size of the `.bss` segment is:

$$B = \sum_{i=1}^{n} s_i.$$

1.3.7 The .rodata Segment

The `.rodata` segment contains read-only data, such as constant literals and string constants. This data cannot be modified during program execution. The size of the `.rodata` segment is determined by the size of the constants stored within it.

If a program has n read-only constants, and each constant c_i requires r_i bytes, the total size of the `.rodata` segment is:

$$R = \sum_{i=1}^{n} r_i.$$

1.3.8 Memory Layout Table

To summarize the properties of each memory segment, the following table presents a comparison of the key characteristics of each section in the memory layout:

Segment	Purpose	Growth Direction	Writable?
`.text`	Stores executable instructions	N/A	No
`.data`	Stores initialized global/static variables	N/A	Yes
`.bss`	Stores uninitialized global/static variables	N/A	Yes
`.rodata`	Stores read-only constants	N/A	No
`Heap`	Dynamic memory allocation	Upward	Yes
`Stack`	Function call management, local variables	Downward	Yes

Table 4: Summary of memory segments in a typical program.

1.3.9 Summary

The memory layout in a computer is divided into distinct segments, each with a specific role. The stack and heap are used for dynamic and local memory management, respectively, while the `.text`, `.data`, `.bss`, and `.rodata` segments store the program's instructions, global variables, and constants. Understanding the characteristics of each segment is crucial for efficient memory management and debugging in system-level programming.

1.4 Performance Metrics in Computer Systems

The performance of a computer system can be rigorously measured and analyzed using several key metrics. These metrics provide a quantitative way to evaluate the efficiency of the system in executing tasks. In this section, we will define and interrelate fundamental performance metrics such as response time, throughput, elapsed time, CPU time, instruction count, clock cycles, and discuss performance optimization via Amdahl's Law.

1.4.1 Response Time and Throughput

The *response time* of a system, also referred to as *execution time*, is the time required to complete a single task or process. Mathematically, let T_r denote the response time for a task P:

$$T_r = T_{elapsed}(P),$$

where $T_{elapsed}$ represents the total time elapsed from the start to the completion of task P. This includes not only the time the CPU spends on P, but also the time the task spends waiting for I/O or other resources.

While response time measures how long a single task takes, *throughput* is concerned with the system's overall capacity to process tasks. It is defined as the number of tasks completed per unit of time. Let $X(t)$ represent the throughput over a period of time t, and n be the number of tasks completed during this time. Then throughput is given by:

$$X = \frac{n}{t}.$$

Throughput and response time are inversely related in many systems, with higher throughput often implying lower response time under certain conditions.

1.4.2 Elapsed Time and CPU Time

The total time to execute a task, known as *elapsed time*, includes both the time spent actively executing on the CPU and the time spent waiting for other resources (e.g., I/O). Let $T_{cpu}(P)$ represent the CPU time for task P, and $T_{io}(P)$ represent the time spent waiting for I/O. The total elapsed time $T_{elapsed}(P)$ can be written as:

$$T_{elapsed}(P) = T_{cpu}(P) + T_{io}(P).$$

The *CPU time* is the time during which the CPU is actively executing the instructions of the program. It can be divided into two components: *user CPU time*, T_{user}, which represents the time spent executing the program itself, and *system CPU time*, T_{system}, which accounts for time spent executing system calls and operating system code. The total CPU time for a task P is:

$$T_{cpu}(P) = T_{user}(P) + T_{system}(P).$$

A key factor influencing CPU time is the number of instructions executed and the efficiency of instruction processing, as discussed next.

1.4.3 Instruction Count and Clock Cycles

The time required for a task to complete is dependent on the number of instructions executed and the number of *clock cycles* needed to execute each instruction. Let $I(P)$ represent the *instruction count*—the total number of instructions executed for a task P. The efficiency of instruction execution is typically measured by the *clock cycles per instruction* (CPI), which denotes the average number of clock cycles required to execute one instruction.

The total number of clock cycles C_{total} required to execute a program P is:

$$C_{total}(P) = I(P) \cdot \text{CPI}.$$

The CPU operates at a certain clock frequency f, typically measured in cycles per second (Hz). Given the total number of clock cycles $C_{total}(P)$, the CPU time $T_{cpu}(P)$ for task P can be expressed as:

$$T_{cpu}(P) = \frac{C_{total}(P)}{f} = \frac{I(P) \cdot \text{CPI}}{f}.$$

Here, the total CPU time is directly proportional to the number of instructions, the CPI, and inversely proportional to the clock frequency.

1.4.4 CPU Clock and Its Impact on Performance

The *CPU clock* frequency f determines the rate at which the CPU executes instructions. Each clock cycle represents a discrete time step in which the CPU processes a small unit of work. Let T_{clock} represent the duration of a single clock cycle, which is the reciprocal of the clock frequency:

$$T_{clock} = \frac{1}{f}.$$

Thus, the total time to complete a program P can be expressed in terms of clock cycles and frequency, further emphasizing the critical role of the clock frequency in determining system performance.

1.4.5 Amdahl's Law

In practice, performance improvements often focus on optimizing certain parts of a program or task. However, the potential benefit from such optimization is limited by the fraction of the program that cannot be improved. This limitation is formalized by *Amdahl's Law*, which provides an upper bound on the speedup that can be achieved by enhancing a portion of a program.

Let p be the fraction of the task P that can benefit from improvement, and let k represent the factor by which this portion can be sped up. The overall speedup S is given by:

$$S = \frac{1}{(1-p) + \frac{p}{k}}.$$

In the case where $k \to \infty$ (i.e., the improved part is infinitely fast), the maximum theoretical speedup S_{max} is:

$$S_{max} = \frac{1}{1-p}.$$

Amdahl's Law highlights the diminishing returns of optimization when a significant fraction of the task remains unaffected by the improvement.

1.4.6 Interrelation of Metrics

These performance metrics—response time, throughput, CPU time, instruction count, clock cycles, and the CPU clock—are deeply interconnected. The response time of a program depends directly on its CPU time, which in turn depends on the number of instructions executed and the clock cycles per instruction. Throughput is inversely related to response time, as it reflects the system's ability to process multiple tasks in a given period.

Optimizing system performance often involves improving the CPI (through hardware efficiency), increasing the clock frequency, or reducing the number of instructions in a program. Amdahl's Law serves as a reminder that the overall system speedup is constrained by the fraction of the task that can be optimized.

1.4.7 Summary

The performance of a computer system can be understood through the interplay of several fundamental metrics. Response time, throughput, CPU time, and elapsed time provide different perspectives on performance. These metrics are influenced by the instruction count, clock cycles, and clock frequency, which together determine how efficiently a program is executed. Amdahl's Law further illustrates the limitations of performance improvements when only a portion of the task can be optimized. By considering these metrics in unison, we gain a comprehensive understanding of computer performance.

2 GNU Assembly Syntax

3 Load / Store and Branch Instructions

In modern computer architectures, instruction sets are composed of various types of instructions, each with a specific role in program execution. Among the most fundamental are *load/store instructions* and *branch instructions*, which handle memory access and control flow, respectively. This section provides a formal mathematical description of these instructions and their behavior.

3.1 Load and Store Instructions

The *load* and *store* instructions are responsible for transferring data between memory and the processor's registers. These operations are central to the manipulation of data in modern architectures, particularly in architectures that follow a *load/store* paradigm, such as RISC (Reduced Instruction Set Computing).

3.1.1 Load Instruction

The *load* instruction reads a value from a specified memory address and transfers it to a register. Let M represent the memory space, R the set of registers, and A the memory address space. A load operation can be mathematically described as a function:

$$\text{Load} : A \to R.$$

Given a memory address $a \in A$, the value stored at that address in memory, denoted $M(a)$, is transferred to a register $r \in R$. The load operation can be expressed as:

$$r \leftarrow M(a),$$

where $\leftarrow$ denotes the assignment of the value from memory to the register.

For example, if a represents the memory address and r is a register, the load operation will fetch the value stored at address a and load it into register r:

$$r \leftarrow M(a).$$

The time to complete a load instruction, denoted T_{load}, depends on whether the memory location is in the cache or main memory, and can be expressed as:

$$T_{load} = \begin{cases} T_{cache} & \text{if } a \in \text{cache}, \\ T_{mem} & \text{if } a \notin \text{cache}. \end{cases}$$

Here, T_{cache} is the time to access a value from the cache, and T_{mem} is the time to access a value from main memory.

3.1.2 Store Instruction

The *store* instruction writes a value from a register to a specific memory location. Mathematically, this operation can be viewed as a function:

$$\text{Store} : R \times A \to M,$$

where a value from register $r \in R$ is stored in memory at address $a \in A$. The store operation is expressed as:

$$M(a) \leftarrow r.$$

This operation writes the value from the register r into the memory location at address a. The time to complete a store instruction, denoted T_{store}, is similar to the load time and depends on the cache status of the memory address:

$$T_{store} = \begin{cases} T_{cache} & \text{if } a \in \text{cache}, \\ T_{mem} & \text{if } a \notin \text{cache}. \end{cases}$$

The combination of load and store instructions allows the system to move data between the memory hierarchy and the CPU's register file, enabling efficient data manipulation and storage.

3.2 Branch Instructions

The *branch* instruction is used to alter the control flow of a program based on specific conditions. Branch instructions are critical for implementing loops, conditionals, and jumps within a program. There are two main types of branch instructions: *conditional branches* and *unconditional branches*.

3.2.1 Unconditional Branch

An *unconditional branch* is a direct jump to a specific address in the program. Let PC denote the program counter, which holds the address of the next instruction to execute. An unconditional branch can be mathematically described as:

$$PC \leftarrow \text{target address}.$$

The target address is typically given as an immediate value or calculated based on the current program counter. If the target address is t, the branch operation simply sets the program counter to this value:

$$PC \leftarrow t.$$

Thus, the control flow jumps to the instruction located at address t, bypassing any intermediate instructions.

3.2.2 Conditional Branch

A *conditional branch* occurs when the jump to a target address depends on the evaluation of a condition. Let C denote a condition (e.g., comparison of two register values). The conditional branch instruction is mathematically represented as:

$$PC \leftarrow \begin{cases} t & \text{if } C \text{ is true}, \\ PC + 1 & \text{if } C \text{ is false}. \end{cases}$$

If the condition C evaluates to true, the program counter is set to the target address t (i.e., the branch is taken). Otherwise, the program continues sequentially, with the program counter incremented by 1 to move to the next instruction.

For example, in a typical conditional branch based on the comparison of two registers r_1 and r_2, the branch operation is:

$$PC \leftarrow \begin{cases} t & \text{if } r_1 = r_2, \\ PC + 1 & \text{if } r_1 \neq r_2. \end{cases}$$

The execution time of a branch instruction, denoted T_{branch}, can be affected by factors such as branch prediction and pipeline stalls. If the branch is correctly predicted, the branch time T_{branch} can be minimal, but if a misprediction occurs, the cost includes pipeline flushing, leading to additional time:

$$T_{branch} = T_{predict} + \begin{cases} 0 & \text{if correctly predicted,} \\ T_{flush} & \text{if mispredicted.} \end{cases}$$

3.3 Performance Considerations for Load/Store and Branch Instructions

The performance of a computer system depends heavily on the efficiency of load/store and branch instructions. The load/store instructions directly affect memory access time, while branch instructions impact control flow and instruction pipeline efficiency.

The total execution time of a program that includes n_{load} load instructions, n_{store} store instructions, and n_{branch} branch instructions can be approximated by:

$$T_{total} = \sum_{i=1}^{n_{load}} T_{load,i} + \sum_{j=1}^{n_{store}} T_{store,j} + \sum_{k=1}^{n_{branch}} T_{branch,k}.$$

In systems with pipelines, branch mispredictions and cache misses significantly influence the total time. Optimizing both the memory hierarchy (to minimize load/store delays) and the branch prediction mechanism (to reduce pipeline stalls) is crucial for enhancing system performance.

3.4 Summary

In summary, load and store instructions manage the movement of data between memory and registers, while branch instructions control the flow of program execution. Understanding the mathematical relationships between memory access times, branch prediction, and instruction execution times is essential for optimizing system performance.

References

[1] ARM Holdings, *ARM64 Architecture Reference Manual*, 2020.

[2] ARM Compiler Reference, *Using the ARM64 calling convention*.

[3] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2nd Edition.