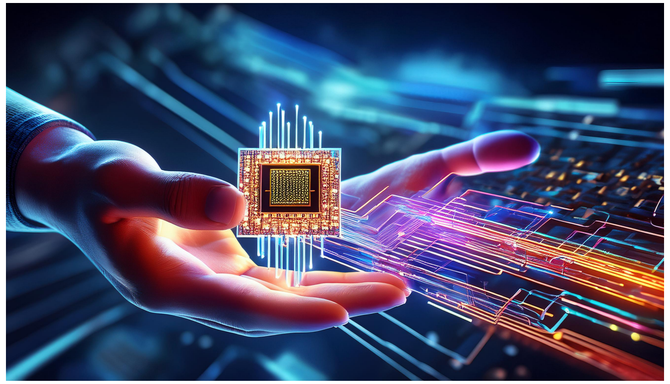
15.10.24 17:56

CSCI 250
Introduction to Computer Organisation
Lecture 3: CPU Architecture II



Jetic Gū

2024 Fall Semester (S3)

Overview

- Architecture: von Neumann
- Textbook: LCD: 9.7; CO: 2.1
- Core Ideas:
    1. ARM Registers
    2. Memory Operations
    3. Lab 3 Part 1: Register Array Implementation

ARM Registers

ARM system registers

- In a normal 32bit ARM CPU, we have 16+1 registers that users can directly access
- R0 - R6: GPR; R8-R10: GPR;
- **R7: Holds system call number**
- **R11 aka FP: Frame Pointer**
- R12-R15 and CPSR: special purpose registers
  - R12: for temporary values, not important to us

P1
ARM Register

R7: System Call Number Reg

- What are system calls?
  - System calls are a security feature
  - System calls separates **user-space instructions** and **privileged instructions**
  - E.g. **changing the system time** is a privileged instruction, it requires a system call to perform

Concept

P1
ARM Register

R7: System Call Number Reg

- What other instructions are protected (privileged)?
  - I/O instructions
  - Context switching
  - Clear/Allocate memory
  - Accessing OS managed resources (filesystems etc.)
  - Basically, anything that can cause your system to crash/freeze, or compromise security

Concept

P1
ARM Register

R7: System Call Number Reg

- Protected instructions cannot be programmed by a normal user, instead, relative functionalities are provided by the Operating System as **System Call functions (**Linux system calls[1)]
- Switching modes:
  - Normally, your programme (like a C user programme) will run in **user-mode**, all instructions you write are executable in user-mode
  - When you need to e.g. access files in the hard drive, you use a **system call function** to do so.
    - The OS switches to **privileged mode** (**kernel mode**)
    - The OS accesses the file for your programme, saves the data at memory addresses that your programme can access, then **exits the kernel mode**
    - Your programme resumes

1. https://man7.org/linux/man-pages/man2/syscalls.2.html

Technical

P1
ARM Register

R7: System Call Number Reg

- Does switching modes compromise performance?
  - A little. Common system call codes can always be stored in the main memory, some very common ones can even be cached. This is managed by the OS
- Are system calls common?
  - Extremely. But it also depends on your choice of programming language and compiler. E.g., a simple `Hello World` could use more than 50 system calls[1]

1. You can use `strace` in Linux to check every system call a programme uses

Technical

P1
ARM Register

R7: System Call Number Reg

- So what does R7 do?
- As seen in the Linux example, system calls are numbered
- R7 helps keep track of which system call is currently being executed
- This helps the OS determine when and how to **get in and out of kernel mode**

Technical

P1
ARM Register

R7: System Call Number Reg

- So what does R7 do?
- As seen in the Linux example, system calls are numbered
- R7 helps keep track of which system call is currently being executed
- This helps the OS determine when and how to **get in and out of kernel mode**
- You will learn a bit more in an OS course. We don’t have to look too hard into these

Technical

P1
ARM Register

R11: Frame Pointer & R13: Stack Pointer

- In short, frames are whole blocks of memory
- By dividing memory regions into blocks, this make it easier to manage access restrictions, control dynamic memory allocation for the operating system
- SP points to the next memory block that is free, allowing for rapid memory allocation
- FP points to the current memory block being used, allowing for rapid memory access
- You will learn a bit more in an OS course. We don’t have to look too hard into these

Technical

P1
ARM Register

R14: Link register

- A register that holds the address to a subroutine’s return
- When you call a function, a subroutine is created. R14 is used to store the return value’s address. Yes, going in and out of a subroutine is also managed by the OS

Technical

P1
ARM Register

R15: Programme Counter

- This is very very important
- A programme counter is where the address of instructions we are executing gets stored
- Specific to ARM, it stores the address of the **Next** instruction to be executed (Think: why is it not the current one?)
- After every CPU cycle, instruction from PC is retrieved, so at the next cycle, we can start execution directly

Technical

Our 16bit ARM thumb CPU

- For simplicity, we'll do things a bit differently from 32bit ARM CPU
- We need only 8 registers, with R7 being PC
- R0-R6 are just normal GPR, no need for specific functionalities
- We also need an Instruction Register, this is outside of the Register Array (not user accessible). We'll discuss this in a future lecture

Memory Operations

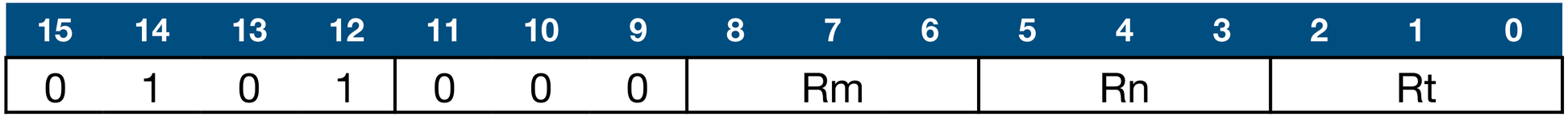
P2
Memory Instruct.

There are 3 Types of Memory Operations in ARM 16bit thumb

- Single Data Item Load/Store
    - A memory address is stored in a register
- Load based on PC address
    - Retrieve from an address relative to Programme Counter
- Multiple Register Load/Store
- For our example, let’s assume register number 7 is PC (since we only have 8 GPR in our 16bit thumb CPU)

P2
Memory Instruct.

Single Data Item Store



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		

- OPa: `0101`, OPb: `000`
  ARM instructions in this case has two parts in its opcode

- `STR Rt, Rn, Rm`

  - Calculates an address from a base register value and an offset, stores a word from a register to memory.

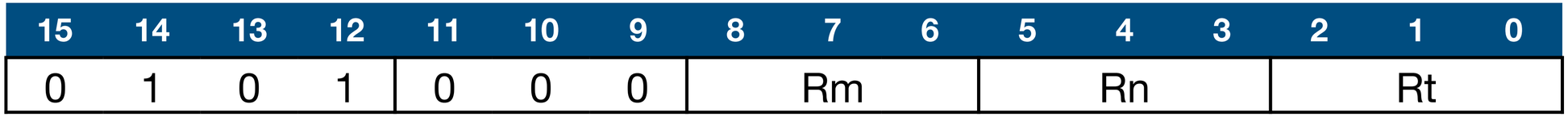
  - Base register: `Rn`; Source register: `Rt`; Offset: `Rm`;

  - Remember, in a 16bit system, a full word is 16bit

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/STR--register-?lang=en

P2
Memory Instruct.

Single Data Item Store



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		

- OPa: `0101`, OPb: `000`
  ARM instructions in this case has two parts in its opcode

- `STR Rt, Rn, Rm`
  - e.g. 0101 000 011 010 000
    OPa OPb Rm Rn Rt
  - Final memory address: `FF03h`
    `(Rn + Rm)`
    Value to be stored to memory: `FF07h`
    `(Rt)`

R7	0000h	*PC
R6	0000h	Normal GPR
R5	0000h	Normal GPR
R4	0000h	Normal GPR
R3	**0003h**	Normal GPR
R2	**FF00h**	Normal GPR
R1	0000h	Normal GPR
R0	**FF07h**	Normal GPR

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/STR--register-?lang=en

Single Data Item Store

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		

- Why offset?
- For example, in C: `int a[100];`
  - C memory allocation is continuous, this allocates 100 words' space in the main memory
    - For byte addressable memory, thats 200 addresses (e.g. `0001h` to `00C8h`)
    - For our case, if our main memory's word length is 16bits, then that's 100 addresses (e.g. from `0001h` to `0064h`, `&a == 0001h`)
  - When you are accessing `a[i]`, you have to store `i` in a register (offset), the address for `a` in another register (base), then access address `a+i`

1. Our system is 16bits, recall in lab 2, the memory you implemented was not byte addressable, each word is 16bit

P2
Memory Instruct.

Single Data Item Store

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		

- An offset makes accessing memory easier, instead of having to waste another CPU cycle on performing offset addition
- The actual use case is far more complicated, your Operating System uses Pages to manage virtual memory, individual variables' address are also relative to a page address, making offset even more useful

Technical

P2
Memory Instruct.

Single Data Item Load

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rm			Rn			Rt		

- OPa: `0101`, OPb: `100`
  ARM instructions in this case has two parts in its opcode

- `LDR Rt, Rn, Rm`

  - Calculates an address from a base register value and an offset, loads a word from a memory to register.

  - Base register: `Rn`; Target register: `Rt`; Offset: `Rm`;

  - Notice the roles of register arguments are different from `STR`

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/LDR--register--Thumb-?lang=en

P2
Memory Instruct.

Single Data Item Store with Immediate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Immediate					Rn			Rt		

- OPa: `0110`, OPb: `0`
  ARM instructions don't necessarily always have the same number of bits for Opcode

- `STR Immediate, Rn, Rt`

  - Calculates an address from a base register value and an offset, stores a word from a register to memory. The offset is a 5bit Immediate value

  - Base register: `Rn`; Source register: `Rt`; Offset: `Immediate`;

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/STR--immediate--Thumb-?lang=en

Technical

P2
Memory Instruct.

Single Data Item Store with Immediate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Immediate					Rn			Rt		

- OPa: `0110`, OPb: `0`
  ARM instructions don't necessarily always have the same number of bits for Opcode

- `STR Immediate, Rn, Rt`
  - e.g. 0110 0 00100 010 000
    OPa OPb Imm5 Rn Rt
  - Final memory address: `FF04h`
    `(Rn + $(Imm5))`
    Value to be stored to memory: `FF07h`
    `(Rt)`

R7	0000h	*PC
R6	0000h	Normal GPR
R5	0000h	Normal GPR
R4	0000h	Normal GPR
R3	0000h	Normal GPR
R2	**FF00h**	Normal GPR
R1	0000h	Normal GPR
R0	**FF07h**	Normal GPR

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/STR--immediate--Thumb-?lang=en

P2
Memory Instruct.

Single Data Item Store with Immediate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Immediate					Rn			Rt		

- Why Immediate?
- For example, in C: `int a, b;`
  - In this case, the memory address for `a` and `b` could be fixed, either in a virtual memory page or just on physical memory.
  - In reality, a computer is incapable of distinguishing labels `a` and `b`, instead an offset relative to the data segment address of that programme is used

1. Our system is 16bits, recall in lab 2, the memory you implemented was not byte addressable, each word is 16bit

Technical

Single Data Item Store with Immediate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Immediate					Rn			Rt		

- The **Instruction Segment** of a programme in the main memory stores the **binary instructions of this programme**. This is usually **read-only**, as programmes don't change their own code on the fly
  - This is often cached or queued*
- The **Data Segment** of a programme contains variables, data structures, etc.

1. Depends on CPU implementation, for our ARM implementation, we don't need to worry about this

P2
Memory Instruct.

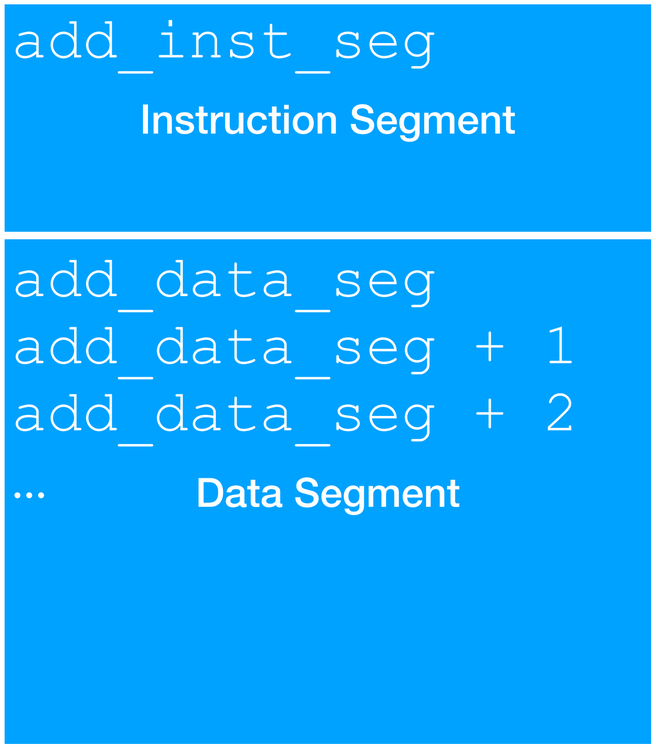
Single Data Item Store with Immediate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Immediate					Rn			Rt		

- Example 1: you have a C programme, in the main function, you declared: `int a=0, b=0;`
  - In this case, two slots in the data segments are created, and allocated to a and b separately
  - Their addresses

    ```
    &a == add_data_seg + 0;
    &b == add_data_seg + 1;
    ```

  - You can access these variables faster using immediate store and load, where `imm5` can be `0` or `1`



Technical

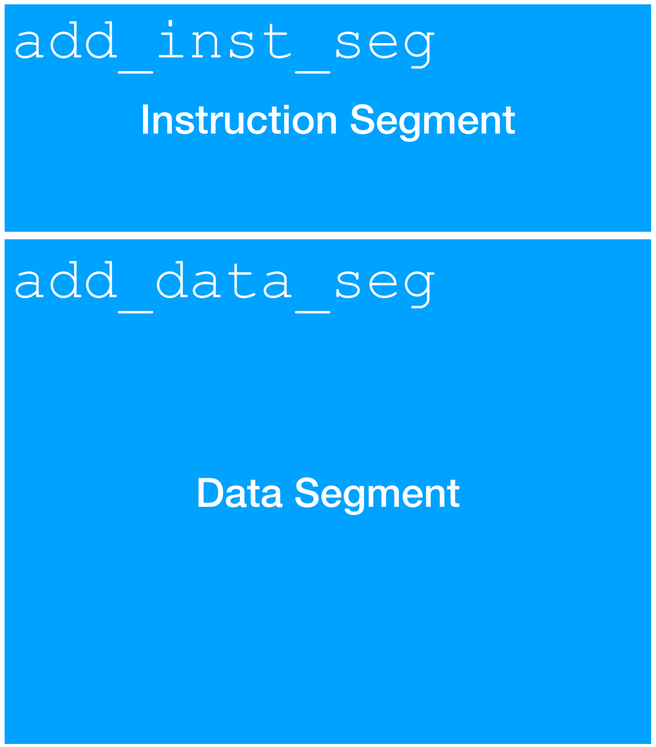
Single Data Item Store with Immediate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Immediate					Rn			Rt		

- Example 2: C++ class and C struct
  - Both C and C++ allocates a fixed amount of memory for each class or struct object/instance
  - For example:

```
struct boo {
    int a;
    int b;
} q;
```

  - `q.a` and `q.b` will have memory addresses like[1] `&q` and `&q + 1`



1. `gcc/clang` has some weird techniques that may cause some differences across different versions and optimisation levels

P2
Memory Instruct.

Single Data Item Load with Immediate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	Immediate					Rn			Rt		

- OPa: `0110`, OPb: `1`
  OPb controls whether it's load or store

- `LDR Immediate, Rn, Rm`

  - Calculates an address from a base register value and an offset, loads a word from memory to register.

  - Base register: `Rn`; Target register: `Rt`; Offset: `Rm`;

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/STR--immediate--Thumb-?lang=en

Technical

P2
Memory Instruct.

Single Data Item Load/Store

opA	opB	Instructions
0101	000	**Store register (STR)**
	001 - 011	Store Half-Word (32bit system), Store Byte, Signed Byte[1]
	100	**Load register (LDR)**
	101 - 111	Load Half-Word (32bit system), Store Byte, Signed Byte[1]
0110	0xx	Store register (STR, Immediate)
	1xx	Load register (LDR, Immediate)
0111, 1000, 1001	-	Store/Load Byte, Halfword[1], SP[2]

1. We do not implement these
2. SP stands for Stack pointer, a special register that we do not use in our implementation

Technical

P3
Lab 3 P1

Lab 3

P3
Lab 3 P1

Implementation of a Register Array

- Part 1:
  - Implement a 16bit Register Cell (CSCI150)
  - Implement 8-to-1 16bit Multiplexers (VHDL, we'll talk about it this Friday)
  - Implementing the Register Array (Circuit diagram)
- Part 2:
  - Implement an ALU (details to be discussed on Friday)
- Due end of next week, or later if necessary. Don't worry we have time.

Concept