ARM Architecture

Julien Freche

julien.freche@lse.epita.fr
http://www.lse.epita.fr/

Outline I

Introduction

Advanced RISC Machine

ARM is for Advanced RISC Machines.

- The most widely used 32-bit instruction set architecture in numbers produced.
- Most used architecture for mobile devices (smartphone, tablets, ...).
- Some time used in desktop environnement,
- Architecture of the Rasberry Pi, PandaBoard, BeagleBoard and many others.

Global overview

Basics

RISC Architecture

Global informations:

- Current version: ARM v7
- RISC (Reduced Instruction Set Computer)
- Bi-endian (but little endian as default)
- 32 bits (but 64 bits with ARM v8).
- Support of multiple CPU modes with some specific registers (User mode, System mode, Interrupt mode, ...)

I will talk about ARM v6 in this talk.

Registers

Registers

ARM v6 has 31 general-purpose 32-bit registers. At any one time, 16 of these registers are visible. The other registers are used to speed up exception processing.

Registers across CPU modes

usr	sys	svc	abt	und	irq	fiq
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_fiq
R9						R9_fiq
R10						R10_fiq
R11						R11_fiq
R12						R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

- R13 is also referred to as SP, the Stack Pointer
- R14 is also referred to as LR, the Link Register
- R15 is also referred to as PC, the Program Counter

Instruction set

Instructions

- Instructions are 32 bits wide
- About 150 instructions
- All instructions are conditionnal
- Three data types: Byte (8 bits), Half-Word (16 bits), Word (32 bits)

Instructions types

There is different instruction types:

- Branch
- Data processing
- Status register transfer
- Load and Store
- Coprocessors
- Exception generating insctrutions

Branch

When a branch occurs, we can optionally:

- Save PC to LR
- Switch to Thumb Mode or Jazelle Mode

Exemple:

```
loop: B loop ; while (1)
```

We can also change the program execution flow by moving a value directly to PC.

Data Processing

Data processing instructions can be split into:

- Arithmetic and logic
- Comparison
- SIMD (Single Instruction Multiple Data)
- Multiply Instructions
- Misc

Exemple:

```
ADD R1, R1, #1 ; Increment R1
```

The right operand can either be an immediate, a register or a shifted register or immediate.
Exemple:

```
MOV R2, R0, LSL #2
; Shift R0 left by 2, write to R2
```

Status Register Transfer

These instructions are used to:

- Set or get values of flags
- Change processor mode
- Change endianess

Exemple:

```
MRS R0, CPSR ; Read the CPSR
```

Load and Store

We use these instructions to load a value from memory or to store a value in memory. We can also optionally:

- Increment or decrement the base register
- Load and store multiple value

Support for Double Word, Word, Half Word and byte
Exemple:

```
LDR R3, [R9], #4
; Load R3 from R9, then R9 = R9 + 4
```

Coprocessors

We use these instructions to tell a coprocessor to perform an operation, this operation can be:

- Data Processing Instruction
- Data Transfer Instruction
- Register Transfer Instruction

Exemple:

```
MCR p14, 1, R7, c7, c12, 6
; ARM register transfer to Coproc 14
; opcode 1 = 1, opcode 2 = 6
; ARM source register = R7
; coproc dest registers are 7 and 12
```

Exception Generating

These instructions are designed to cause an exception:

- Software interrupt instructions
- Software breakpoint instructions

Exemple:

```
SWI <immediate_value> ; Software Interrupt
```

Extensions

Extensions

Some extensions:

- Thumb: a new set of 16 bit wide instructions to reduce program size. But instructions are not as powerful as original instructions.
- Jazelle: allow execution of Java Byte code directly on the processor.
- VFP: operations on float point numbers.
- NEON: Advanced SIMD instructions.

Basics of emulator writing

Main Loop

Introduction

An emulator is simply a program that will behave exactly as the processor. This program will understand all instructions supported by the processor .
In this section we will talk about a way to code an emulator.

Main loop

The main loop of an simple emulator can be written as the following.

```
int main()
{
  Cpu cpu;
  Memory mem;

  while (1)
  {
    uint32_t current_inst;
    current_inst = mem.fetch(cpu.get_pc());
    cpu.decode(current_inst);
    cpu.execute();
  }
}
```

The main loop includes three stages: fetch, decode and execute.

Fetch

Fetch

This stage is may be the easiest, we simply need to get the instruction pointed to by the PC register.

```
class Memory
{
  public:
  uint32_t Memory::fetch(uint32_t pc)
  {
    return mem_[pc];
  }

  private:
  std::array<uint32_t, N> mem_;
};
```

We suppose that mem_ is large enough to store all words that will be used by the processor. This solution is simple but ugly... the processor can potentially use several Gb of memory.

Fetch

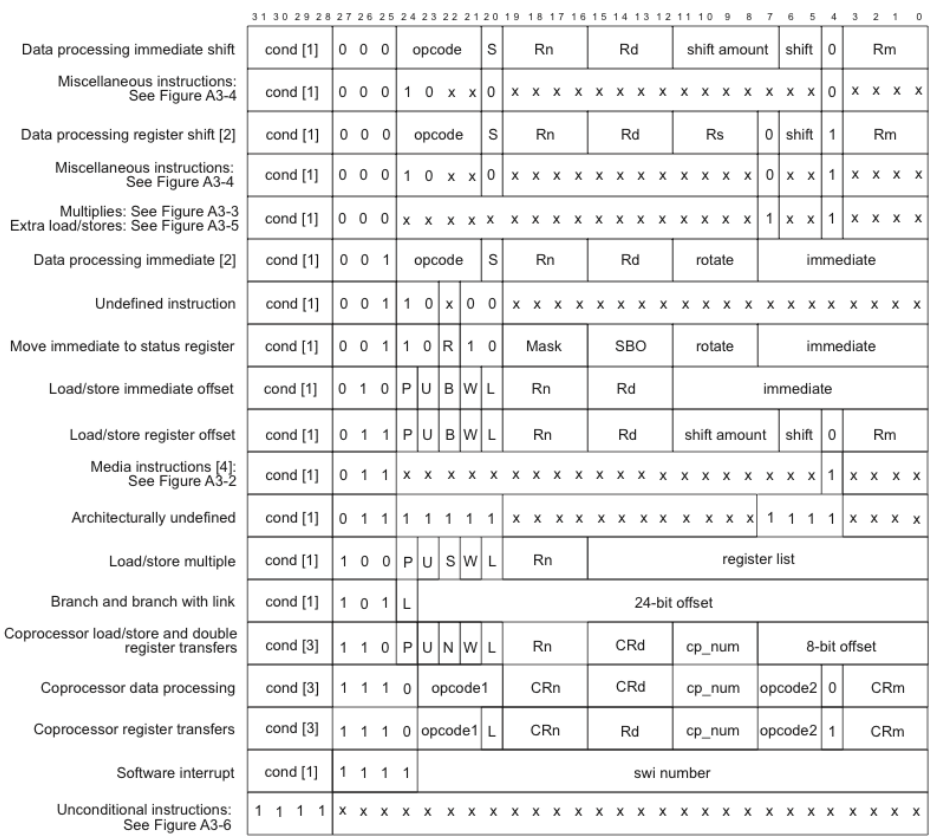
A better solution is may be to use an unordered_map.

- We create a relation between adress of the emulated CPU and adress of the process.
- Use the right amount of memory
- If the processor want to use more memory, the emulator can call mmap to obtain more memory from the OS.

Decode

Decode

This stage is not very complex to unterstand but not very pleasant to code.
The principe is simple, we have to decode all fields of the instruction to unterstand what the emulator will have to do.



Decoding a SWI

A simple exemple is the SWI instruction:

```
void decode_swi(uint32_t inst)
{
  uint8_t cond = extract(inst, 28, 32);
  uint8_t swi_code = extract(inst, 25, 28);
  uint32_t swi_nb = extract(inst, 0, 25);

  // Make use of these variables.
}
```

We suppose we have an extract function that will extract some bits in the instruction (it's just a play with bit mask).

This exemple is simple but the decode is sometimes hard with more complex instruction (Data Processing with register shift, Load with post-incrementaion, ...)

Execute

Principle

In this stage, we have to execute the instruction previously decoded.

- Execution of instruction is very dependant of the type of the instruction
- We have to be carreful about flags update
- The result of the instruction may depend of the CPU mode.

All instructions are conditionnal, so we will have to code functions that returns true if the condition is verified.

A way to do it is to use C++11 Lambda functions.

Conditionnal instructions and Lambda Functions

```
# define EQ 0x0
// ...
# define AL 0xe
# define XX 0xf

typedef bool (*condition)(const cpu&);

class Instruction
{
  public:
    Instruction();
    bool cond_is_true(const Cpu& cpu,
                      uint32_t inst) const;

  private:
    std::array<condition, 16> cond_array_;
};
```

Conditionnal instructions and Lambda Functions

```
Instruction::Instruction()
{
  cond_array_[EQ] = [](const Cpu& cpu)
  {
    return cpu.get_status_z();
  };
  //  ...
  cond_array_[AL] = [](const Cpu&)
  {
    return true;
  };
}

bool Instruction::
     cond_is_true(const Cpu& cpu, uint32_t inst) const
{
  uint8_t cond = extract(inst, 28, 32);

  return ((cond_array_[cond])(cpu));
}
```

Dynamic Binary Translation

Principle

Principle

Let's say I have an ARM executable file and my computer is running on a x86. It would be nice to execute it transparantly.

- Dynamic Binary Translation will translate dynamically instructions in ARM to x86.
- Translated blocks can be executed directly by the processor.
- This principle is used for exemple in Rosetta, a OS X tool to execute a binary with PPC instructions.
- It's different of a 'sandboxed emulation', syscalls are handled by the host kernel.

Instruction translation

Instructions

With dynamic translation we translate instruction only when needed and only one time.

- When a function call is performed, we can translate the function and store it for further use.
- Avoid to translate unused blocks.
- If translatation is fast enough, the translated program has similar performance compared to a native binary.
- Work only if all instructions can be translated

Memory translation

Memory translation

The binary file contains informations about the memory layout of the program, we have to respect it.

- If the program use only relative jump, it's possible to change its position in memory.
- If we need to change its position in memory (for whatever reason), we can add an offset to the right instructions when translating.
- Another solution is may be to create a map to translate real addresses to required adresses.

Syscalls

Syscalls

The translated program probably want to interract with the kernel, but it could be a problem.

- We have to pay attention to the call convention, arguments have to be in the right register.
- Syscall numbers have to be the same between the binary to translate and the host. If not we can create a translation table.
- If host is running on 64 bits, it will have to support 32 bits syscalls.

Questions ?