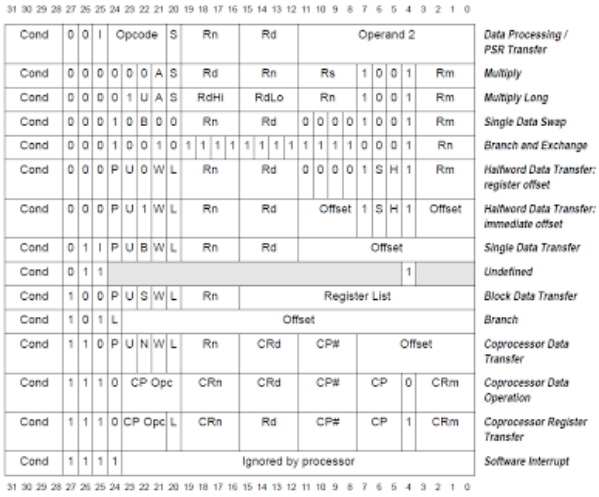
The art of emulation

Sunday, 5 September 2010

Decoding the ARM instruction set

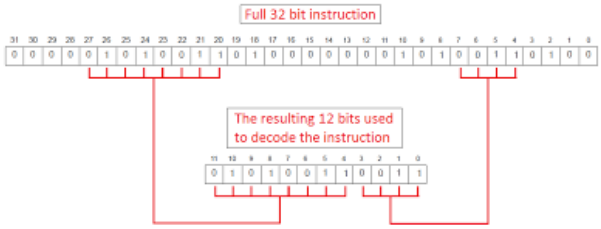
I haven't got far myself in decoding the ARM instruction set, but I have the basic idea, so I'll share it with anyone who's remotely interested. In writing this, I'm assuming you have a base knowledge about the ARM processor and you have studied the instruction sets. This will only just scratch the surface of the ARM instruction set, basically, it will give you a logical pattern to follow that can be used to decode the entire instruction set.

Unlike THUMB instructions, each ARM instruction is a full 32 bits, so decoding each instruction is a bit more complex, but not that much. First of all, take a look at the binary opcode format for an ARM instruction.



This reference isn't completely accurate though, it does miss some important details which we will need to decode the first instruction and which I'll go over soon.

So, to start off with, when we decoded the THUMB instruction set, we were simply using the upper 8 bits of the instruction, which allowed us to unambiguously decode the instruction. This time however, things are a little harder... let's say we took bits 27-20, and tried to figure out which instruction it is, this would be ambiguous, as more than one type of instruction can easilly have all of bits 27-20 unset (0). So, what we are going to do is take the high 8 bits (bits 27-20) and the base 4 bits (bits 7-4), and add them end to end. Here's a picture that should demonstrate the principle a little better.

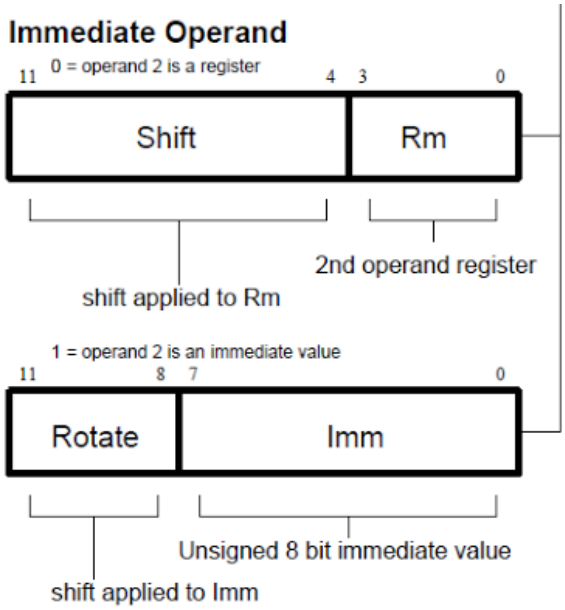


In C/C++, to actually retrieve this value, you just need to to a bit of bitwise finicking. For example.

```
u32 instruction = fetch();
u16 _12Bits = (((instruction >> 16) & 0xFF0) | ((instruction >> 4) & 0x0F));
```

So now we have a 12 bit number that we are going to use to decode the instruction. Let's start from the beginning.

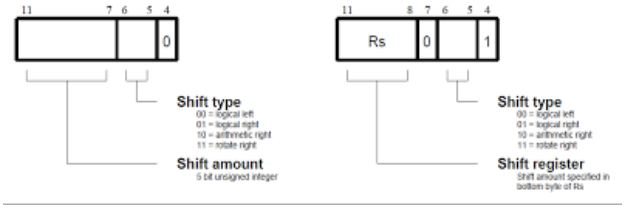
What if this 12 bit number was 0, ie every bit was unset. By looking at the opcode format, you can see that there is only 1 instruction format that allows this, the "Data processing / PSR" instructions. Now, already we know what type of instruction this is, but we need more info to see exactly what instruction it is. In the "Data processing / PSR" format, you see an "Operand 2" field. This field has a format that is not shown in the binary opcode format above, here is the specifications for the operand 2 field.



In our circumstance, the "immediate" bit (bit 25) is not set (all bits are zero, remember), so that means that we are using a shifted register. (I wont go into barrel shifts at the moment, this is beyond the scope of this.... you know the drill...). Here is a more formal way of putting it, straight from the manual.

> "When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction."

The shift field format is as follows...



I know this is all very complicated, but it is best that you read the manual to get a better grasp of what the hell I'm going on about.

Anyway, stay with me now! Let's go through what we have gathered about this instruction so far... assuming that all our 12 bits are 0 that is...

1. **This is a Data Processing instruction.**
2. **It uses a shifted register barrel shift operation.**
3. **It is an AND instruction (opcode is bits 24-21).**
4. **Bit 4 is 0, so this must be a shift operation that shifts the shifted register by an immediate value, hence, uses the first shift field format (format to the right of the above picture).**
5. **The entire shifted register field is 0, so this shift operation must be a**
6. **This does your head in if you think about it too much...**

So, by process of elimination, this instruction must be... *drumroll*

AND Rd, Rn, Rm, LSL #imm5

This instruction does the following.

- **Logically shift Rm left (LSL) by an immediate 5 bit value.**
- **Logically AND Rn with Rm**
- **Store the result in Rd**

Anyway, to conclude, I hope this was informative in some way, even if you didn't know what I was rambling on about. Next time you look at a GBA or GBA emulator, think about what's going on inside, it's like "Whoaaa"... imo

EDIT: Forgot to mention the condition codes...
Every ARM instruction can be executed conditionally, and bits 28 to 31 specify the condition that this instruction executes under. For more info, check out the manual, no point in me making is even harder to understand than it already is.

Peace

Posted by Ryan at 23:17

7 comments:

r0cksTar 6 September 2010 at 01:29

nice blog, check mine bro

http://no-pulse-blog.blogspot.com/

Reply

applecider 6 September 2010 at 01:49

man, I wish I was as good with computers as you. gj!

just doin daily run.
herp.derp.

Reply

Sup 6 September 2010 at 04:18

Supportin!

Reply

Johnn 6 September 2010 at 05:28

Jesus Christ, do you write all this yourself?

Reply

Ryan 6 September 2010 at 07:39

Johnn, yes I did write all this myself, and to be honest, it was as much for my benefit as it was for other peoples. Writing things down like this helps get them clear in my mind.
Thanks for everyones support, I really appreciated it. :)

Reply

RobertoTheBlade 6 September 2010 at 10:52

awesome blog man

Reply

A musician and gamer 6 September 2010 at 11:38

thanks for that

http://thatguysstuff.blogspot.com/

Reply

Enter your comment...

Comment as: Matan Lurey (Gc

Sign out

Publish Preview

Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)