PinKY Reference: EE480 Advanced Computer Architecture

Instruction set design is hard. Prof. Dietz has designed dozens of instruction sets in the three decades he's been a professor, and it still isn't easy for him to get things right. Thus, rather than giving you complete freedom to design your own instruction set, we're going to walk through the design logic for a reasonably well-crafted one that he built specifically for Fall 2018 EE480. However, this design is not complete -- each student must devise their own encoding of the instructions implement their own assembler.

PinKY Overview

PinKY is a somewhat strained acronym for PINkie from KentuckY. PINkie? Well, there's ARM, then there is the Thumb subset, and now there's PinKY. If instead it reminds you of this, well, that's OK too. The key point is that it is a very simple little architecture with a variety of similarities to ARM.

Before we get started on this, let's just be completely clear: ARM is not a simple instruction set. Want proof? Here is a six-page "reference card." The instructions are fixed length, but encoding is a mess. So, PinKY isn't really an ARM subset, but more a simple design having some ARM-like features and using ARM assembly language constructs where that isn't awkward.

In any case, PinKY is a 16-bit machine. Everything is 16 bits wide: instructions, addresses, data -- even floating-point data. It isn't even byte-addressed for memory. In fact, a little C-subset compiler for PinKY might even treat `char`, `short`, `int`, and `float` as all being 16 bits long. Beyond the desire to greatly simplify everything, the 16-bit encoding is the major source of compromises in PinKY.

PinKY Conditionals

Perhaps the most distinctive aspect of ARM instructions is the extensive use of condition codes, including allowing their use as predicates to conditionally execute instructions. Giving a condition name as a suffix of the instruction name makes the instruction execute only if the given condition is true. There is also an additional suffix, `S`, that specifies the marked instruction should set condition codes. *These suffixes can be applied to any instruction*. In PinKY, the condition code structure is simplified to just four suffixes:

Suffix	Meaning	Example
none	Unconditionally execute; Zero flag is not altered	`ADD`

S	Unconditionally execute and set condition; the Zero flag is set to (result==0)	ADDS
EQ	Execute only if Zero==1; Zero flag is not altered	ADDEQ
NE	Execute only if Zero==0; Zero flag is not altered	ADDNE

PinKY General-Purpose Registers

PinKY essentially has the same 16 registers seen in user-mode ARM, but each is only 16 bits wide, not 32 bits. The registers are named in the obvious way, as `r0` through `r15`. However, there are also special names for a few registers, but this isn't like MIPS -- register 0 isn't special.

`r13`
The stack pointer, `sp`, which points below the last valid entry on the stack
`r14`
The link register, `lr`, which holds the return address
`r15`

The program counter, `pc`

You can reference any of these registers by number instead of by name. However, here is an AIK specification of the names:

```
.const {r0     r1   r2   r3   r4   r5   r6   r7 r8     r9   r10
r11  r12  r13  r14  r15 13    sp   14   lr   15   pc}
```

PinKY Operand2

The ARM instruction set allows Operand2 to be either a register or a constant immediate value. Well, so does PinKY. However, because each PinKY instruction is only 16 bits long (not 32), there just isn't space for a big constant. Thus, PinKY does something funky: it uses a modal instruction. Actually, ARM is unusual in having some modal instructions of it's own, so maybe this isn't so unnatural for an ARM-like processor after all?

A register operand can be specified in the obvious way -- by naming the register. For example, `r13`, `sp`, or even `9+4`, would mean register 13. A constant operand is specified using a # character as a prefix. For example, `#13`, or `#9+4`, would represent the constant 13. The catch is: **constants are normally sign-extended 4-bit vaues**, which means that you can have values from -8 to +7. Sadly, 13 isn't in that range. So, how does this work?

If the constant specified is in range, it is in fact to be encoded as a 4-bit number in the instruction. However, if it does not fit, the top 12 bits are to be encoded in a `PRE` instruction. In other words, and instruction written as:

```
ADD r1,#0x1234
```

Would actually be encoded as the two-instruction sequence:

```
PRE #0x123
ADD r1,#0x4
```

The `PRE` instruction simply overrides the usual sign extension for the next instruction with a constant Operand2. Yeah, it's weird, but it is also very flexible. Incidentally, yes, you can have `PREEQ` and `PRENE`. However, it doesn't make sense to apply the `S` suffix to `PRE`.

PinKY Instructions

The ARM instruction set is probably the least RISCy RISC instruction set ever created. In fact, it is a complex mess. Yes, I did just say that in writing. It's a fact -- especially when you include Thumb and Jazelle. Well, PinKY is designed to be a lot less messy. Unfortunately, that means that some ARM features, such as the ability to have shift/rotate of an operand to an instruction, just aren't in PinKY. will often take more PinKY instructions to do anything. Despite that, PinKY does feel like ARM. For example, not only can all instructions be executed conditionally, but every instruction allows a constant instead of a register for the second operand.

Instruction	Description	Functionality	Suffix Forms
`ADD` *`Rd, Op2`*	ADD integers	*`Rd += Op2`*	ADD, ADDS, ADDEQ, ADDNE
`ADDF` *`Rd, Op2`*	ADD Floats	*`Rd += Op2`*	ADDF, ADDFS, ADDFEQ, ADDFNE
`AND` *`Rd, Op2`*	Bitwise AND integers	*`Rd &= Op2`*	AND, ANDS, ANDEQ, ANDNE
`BIC` *`Rd, Op2`*	BItwise Clear integers	*`Rd &= ~Op2`*	BIC, BICS, BICEQ, BICNE
`EOR` *`Rd, Op2`*	Bitwise Exclusive OR integers	*`Rd ^= Op2`*	EOR, EORS, EOREQ, EORNE
`FTOI` *`Rd, Op2`*	Convert Float TO Integer	*`Rd`* `= ((int)`*`Op2`*`)`	FTOI, FTOIS, FTOIEQ, FTOINE
`ITOF` *`Rd, Op2`*	Convert Integer TO Float	*`Rd`* `= ((float)`*`Op2`*`)`	ITOF, ITOFS, ITOFEQ, ITOFNE
`LDR` *`Rd,`* `[`*`Op2`*`]`	LoaD word into Register	*`Rd`* `= memory[`*`Op2`*`]`	LDR, LDRS, LDREQ, LDRNE
`MOV` *`Rd, Op2`*	MOVe (copy) into register	*`Rd = Op2`*	MOV, MOVS, MOVEQ, MOVNE
`MUL` *`Rd, Op2`*	MULtiply integers	*`Rd *= Op2`*	MUL, MULS, MULEQ, MULNE
`MULF` *`Rd, Op2`*	MULtiply floats	*`Rd *= Op2`*	MULF, MULFS, MULFEQ, MULFNE

NEG *Rd*, *Op2*	NEGate	*Rd* = -*Op2*	NEG, NEGS, NEGEQ, NEGNE
ORR *Rd*, *Op2*	Bitwise OR Register? integers	*Rd* |= *Op2*	ORR, ORRS, ORREQ, ORRNE
PRE #*constant*	Constant PREfix	*prefix* = *constant*	PRE, PREEQ, PRENE
RECF *Rd*, *Op2*	RECiprocal Float	*Rd* = 1.0/*Op2*	RECF, RECFS, RECFEQ, RECFNE
SHA *Rd*, *Op2*	SHift Arithmetic signed integers	*Rd* = ((*Op2*>0) ? *Rd*<<*Op2* : *Rd*>>-*Op2*)	SHA, SHAS, SHAEQ, SHANE
STR *Rd*, [*Op2*]	STore Register	memory[*Op2*] = *Rd*	STR, STRS, STREQ, STRNE
SLT *Rd*, *Op2*	Set Less Than integers	*Rd* = (*Rd*<*Op2*)	SLT, SLTS, SLTEQ, SLTNE
SUB *Rd*, *Op2*	SUBtract integers	*Rd* -= *Op2*	SUB, SUBS, SUBEQ, SUBNE
SUBF *Rd*, *Op2*	SUBtract Floats	*Rd* -= *Op2*	SUBF, SUBFS, SUBFEQ, SUBFNE
SYS	SYStem call	invokes operating system; halts simulation	SYS, SYSEQ, SYSNE

This instruction set is complete enough that I hope to be giving you a compiler (including full C source code) that translates programs written in a little dialect of C into PinKY code. It's not going to be a particularly smart compiler (ok, it's really dumb), but it will show you how PinKY can be used for complete programs.

You might have noticed that the above instruction set doesn't have any control flow instructions, such as branches. Well, actually, it does -- it's called ADD. If you want to branch on equality to location *lab*, you would execute ADDEQ pc,#(*lab*-.). Strange, eh?

Some Encoding Hints

Determining how to encode the above instructions as bit patterns is a key part of your project. However, there are a few rules:

- There are apparently 21 different types of operations and a whopping 82 different instruction names! Well, you're going to need to be a bit clever to encode all that into just 16 bits and your specification of the instruction set would get long. However, think of it this way. The register destination, Rd, needs 4 bits and Op2 needs 4 bit to specify either a register number or a 4-bit constant, plus one more bit to say which Op2 is. That's 9 bits.

So, you have 7 bits in which to encode 82 instructions... which doesn't sound so bad. The catch is that the `PRE` instruction needs a 12-bit operand, so it must be distinguished from all other instructions using at most 4 bits. If you treat the condition code as two bits of the opcode, that means you have to have two bits of the remaining opcode field identify `PRE`....

- You do *not* need to use `.alias` and other "fancy" features of AIK to build your assembler. Writing a separate pattern for each instruction type is fine. However, the instruction set here appears huge: I strongly recommend you use `.alias` to factor-out the condition code stuff. This will also help in that each instruction (well, except `PRE` or `SYS`) really has three different forms for Operand2 independent of condition suffix: a register, a 4-bit sign-extended constant, or a 16-bit constant.
- You'll also want to use the AIK ability to test values. Basically, any constant that doesn't fit in four bits needs to cause a `PRE` instruction to be emitted first. That gets slightly tricky because `ADDEQ r8,#42` basically needs to turn into code that looks like the user wrote `PREEQ #2` followed by `ADDEQ r8,#10`. Perhaps that's more clear as `PREEQ #(42>>4)` followed by `ADDEQ r8,#(42&0x000f)`. Remember, thanks to the sign extension, a constant *c* only fits in the 4-bit field if *((c>-9)&&(c<8))*.

I bet you're also a bit worried about those floating-point values. Well, don't worry. They simply get entered as the appropriate bit patterns, typically in hexadecimal. You also will not need to implement the floating-point arithmetic until the last project.

Change Log

The hope was that everything would be perfect, but I have made one change since announcing PinKY on Sept. 14. As of Sept. 17, the `MVN` instruction has been dropped and `NEG` has been added; both are ARM instructions, but negate is more useful than not.