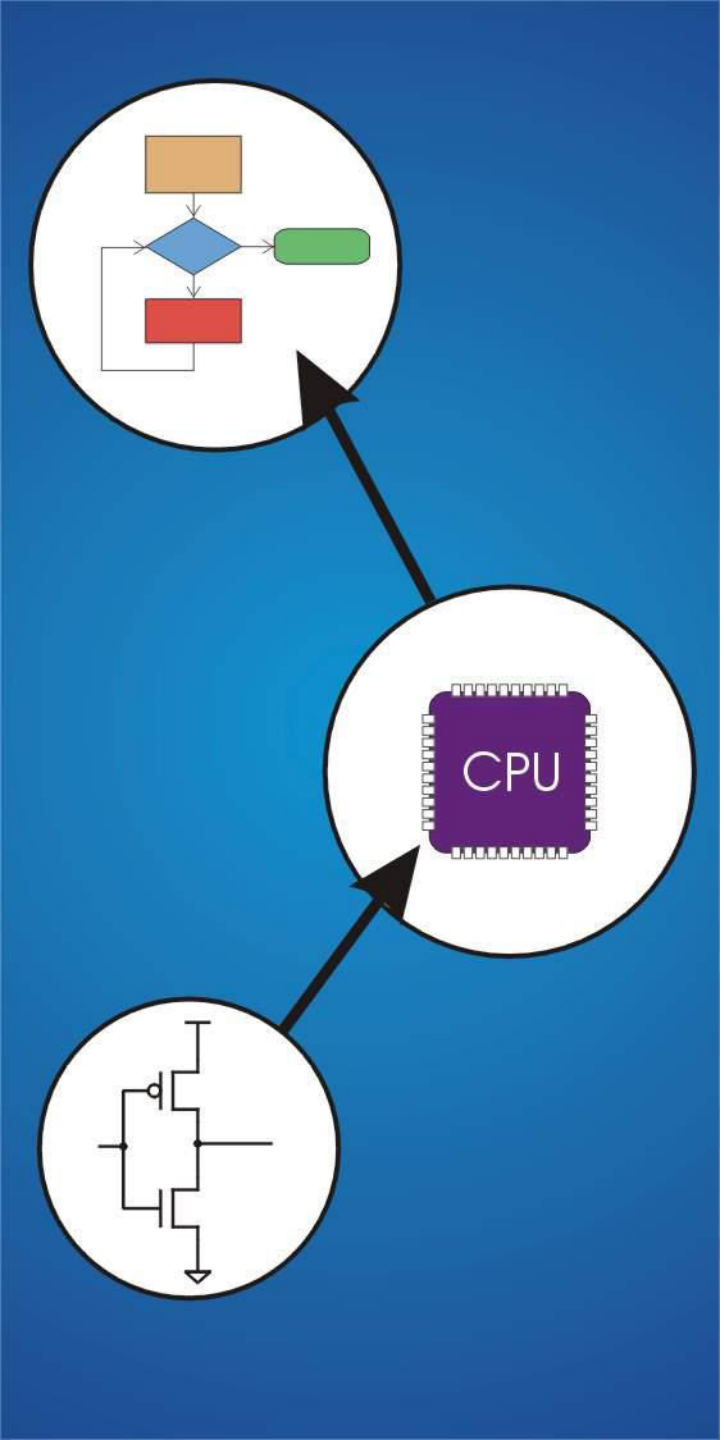


The ARM ISA for LC-3 Programmers

with sources from ARM, Ltd.
Revised William D. Leahy Jr.

Today’s number is 72,000

The claim we made was…

LC-3 is easy to learn

We will teach you LC-3, and *then it will be "easy" to learn another, real ISA*

Time to show you we were right!

A thought: *A new ISA is like a new car. The pedals are there, there's a steering wheel, etc. But the headlight control, high beams, wipers, etc, are all in a slightly different place. Still, you can drive it without too much trouble, and not (hopefully) crash into something!*

REVIEW: ISA= Instruction Set Architecture

ISA = The binary "language" used to talk to a processor. All of the *programmer-visible* components and operations of the computer

- **memory organization**
  - **address space -- how may locations can be addressed?**
  - **addressability -- how many bits per location?**
- **register set**
  - **how many? what size? how are they used?**
- **instruction set**
  - **Operation codes)**
  - **data types (2's complement, floating-point, etc)**
  - **addressing modes.**

Some ARMisms

ARM was a small company in the UK. It was sold to SoftBank in 2016 and is being bought now by NVIDIA.

Technically, they make nothing – they sell the rights to use ARM-compliant processors in your design!

Originally called Acorn Computer Company

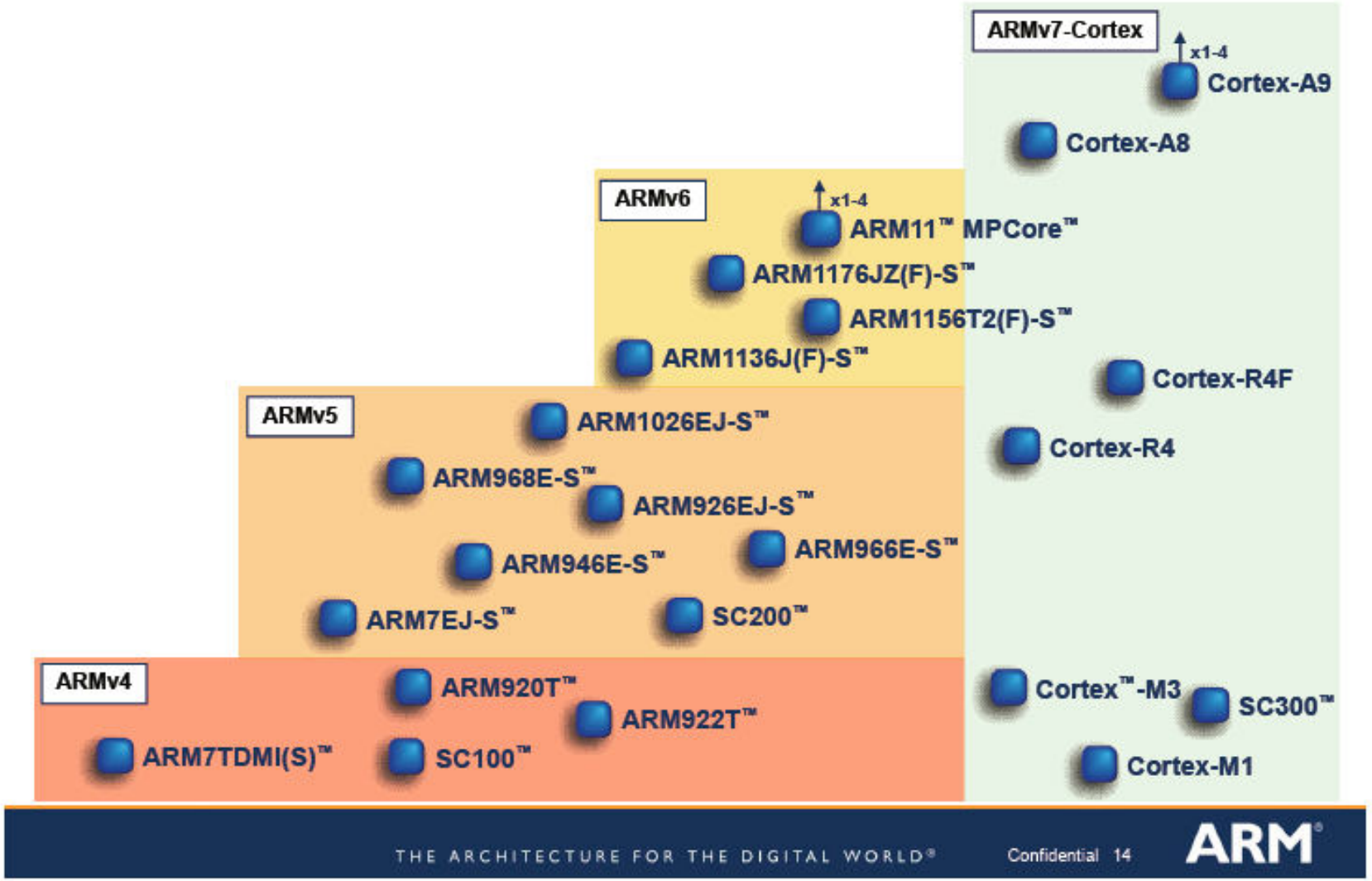
- **The Acorn was a small, cheap computer that used the Mostek 6502**
- **Acorn decided to make their own processor: the Acorn RISC Machine (A.R.M.)**
- **Since that time, A has become "Advanced" instead of "Acorn," and company changed its name to "ARM"**

ARM has had multiple *versions*

The *version* != the processor number

So ARM-7 does NOT use ARM-version7's ISA (it uses ARM version 4)

Architecture Versions



What's the same as LC-3?

ARM and LC-3 are both "load/store" architectures

- **This means that values get into registers using a LOAD instruction and go to memory using a STORE instruction**

ARM and LC-3 both have fixed-format instruction encodings

- **All instructions in LC-3 are 16b long. ARM has two ISAs, in one, they are 32b long (what I'll present), and they are 16b in the other**

Register-register addressing

- **In ARM as in LC-3, many (most) instructions take two source registers and store their results in a destination register**

Address space, addressability

LC-3 Address Space is 2^{16} words, where a word is 16b

- **all locations are word addressable**
- **the address size is 16 bits**

ARM Address Space is 2^{32} bytes

- **all locations are byte-addressable**
- **the address size is 32 bits**

Processor Modes*

User	Normal execution mode
FIQ	High priority (fast) interrupt request
IRQ	General purpose interrupts
Supervisor	Protected mode for O/S
Abort	Used for memory access violations
Undefined	Used to handle undefined instructions
System	Runs privileged O/S tasks

LC-3 just has User and Supervisor modes

*Modes are entered via interrupts or software instructions

The ALU and the registers

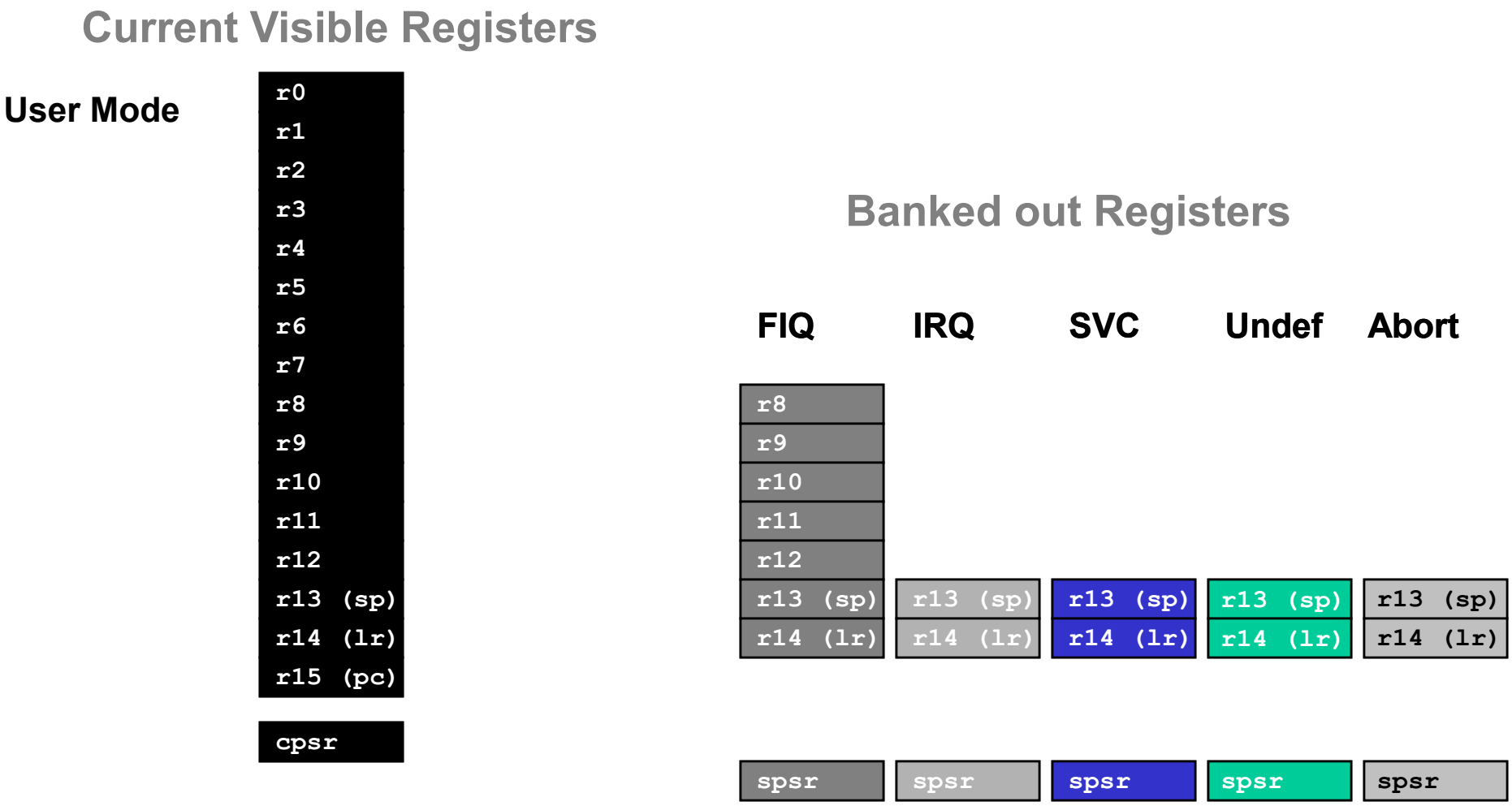
ARM has 16 registers, R0-R15

- **UNLIKE LC-3, *the PC is a register that the programmer can access directly* (PC = R15 in ARM)**
- **All registers are 32b wide**
- **On older ARM ISAs (versions 1-5), storing to PC has "unpredictable results"**

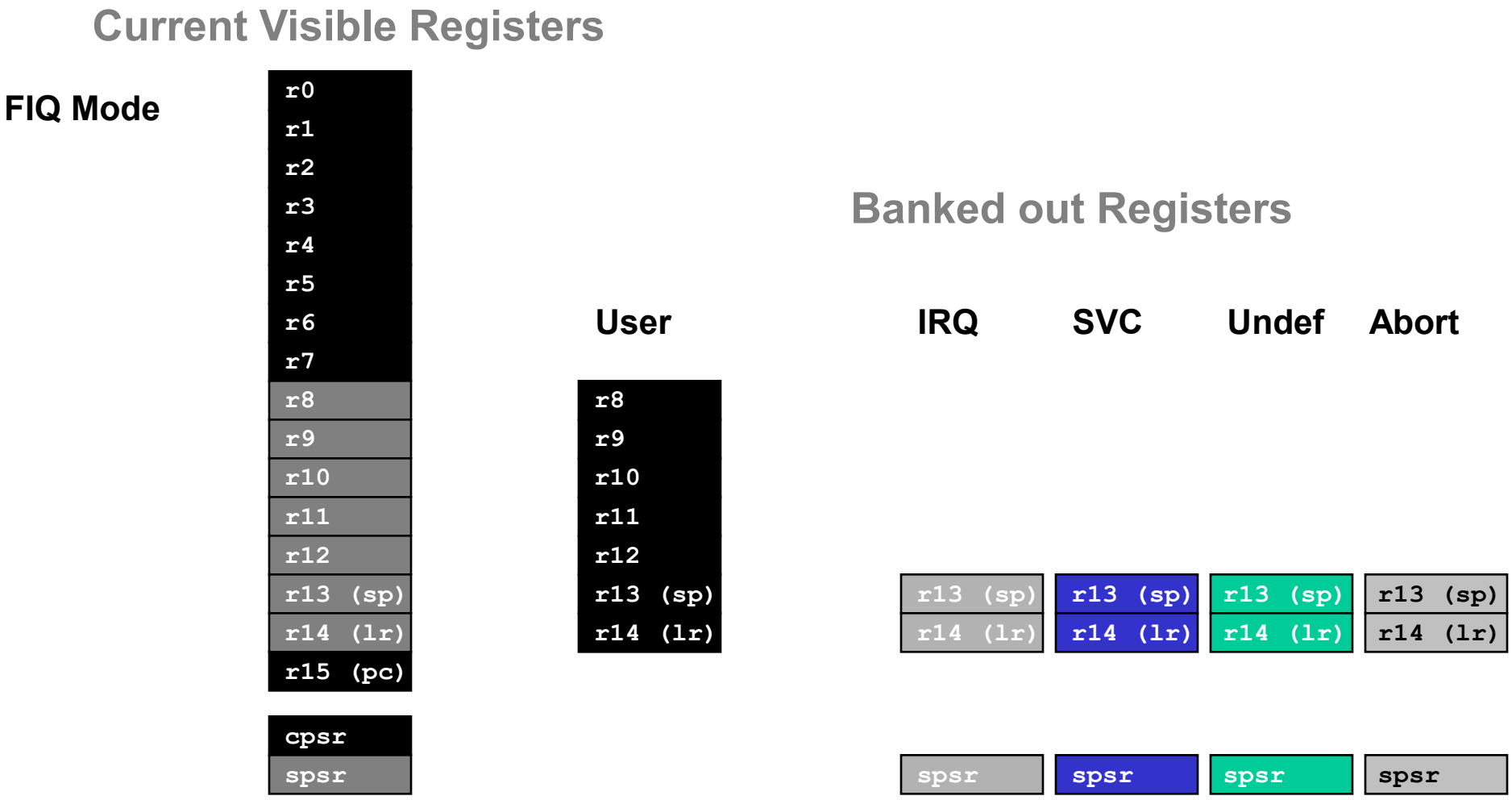
Other special purpose registers:

- **R11 ("FP") is the frame pointer (similar function to R5 in LC-3)**
- **R14 ("LR") is the link register (same function as R7 in LC-3, it holds the return address for a subroutine call)**
- **R13 ("SP") is the stack pointer (same function as R6 in LC-3, it holds the address of the top of stack)**

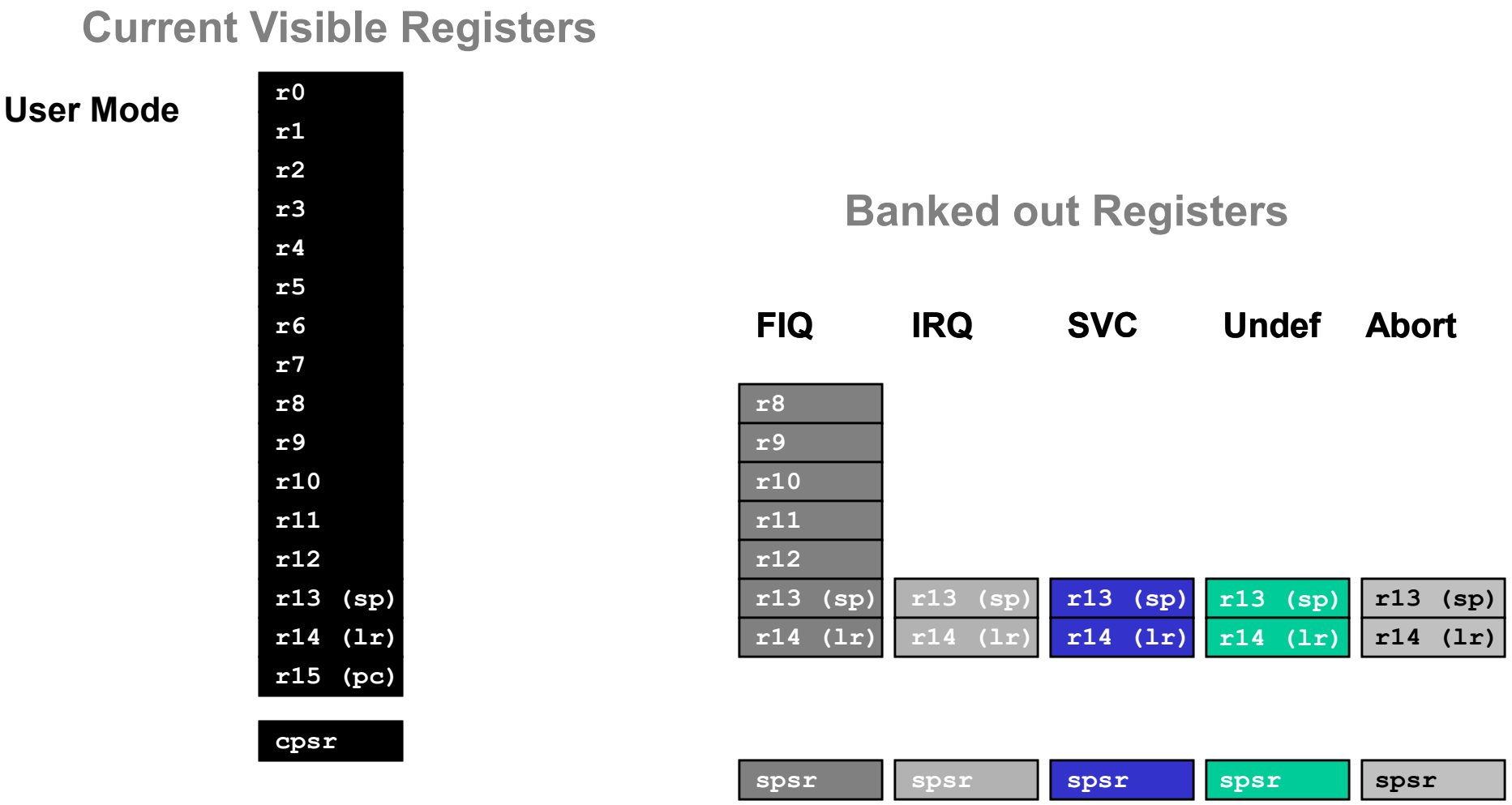
The ARM Register Set: Register Banking



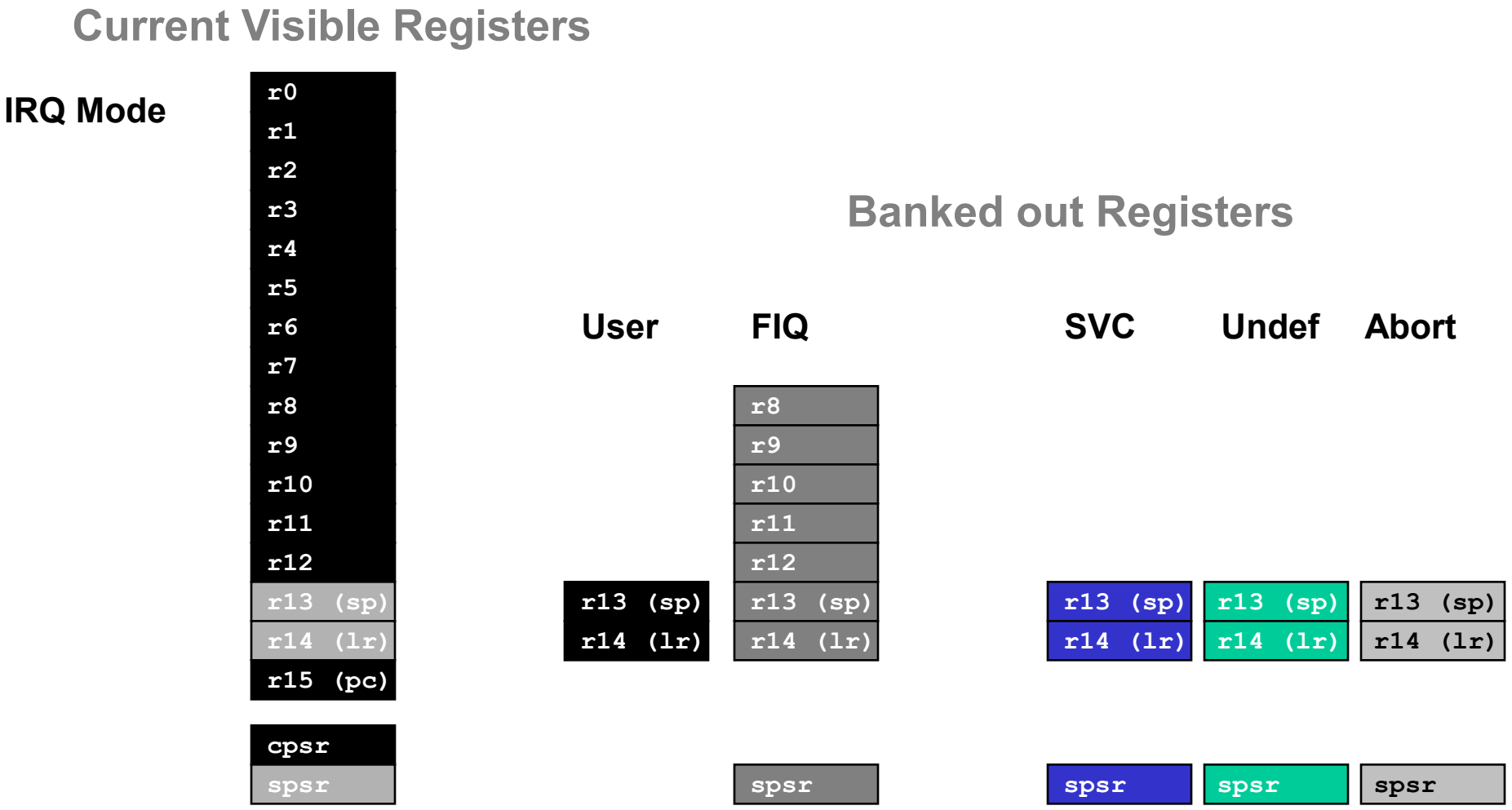
The ARM Register Set



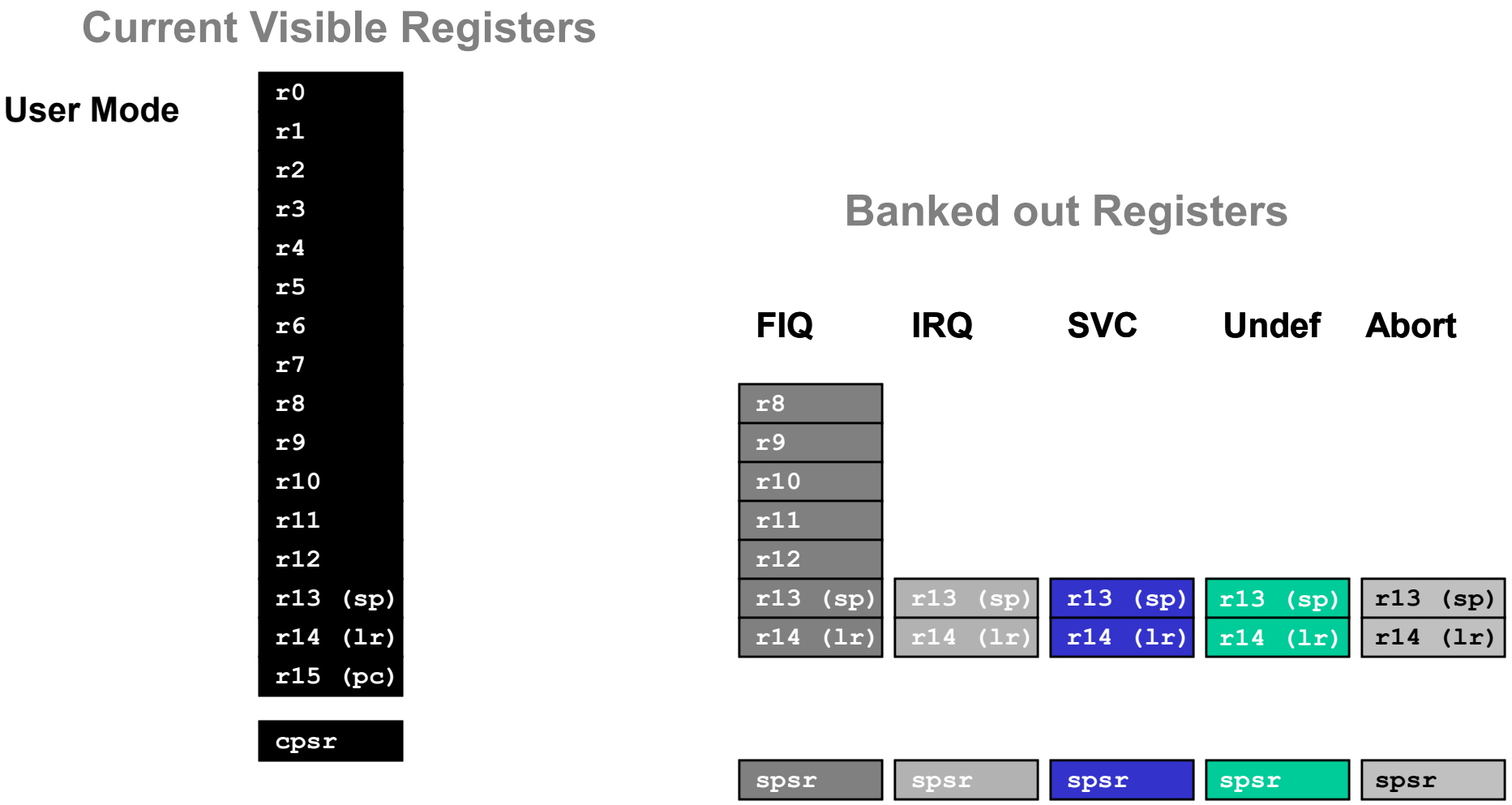
The ARM Register Set



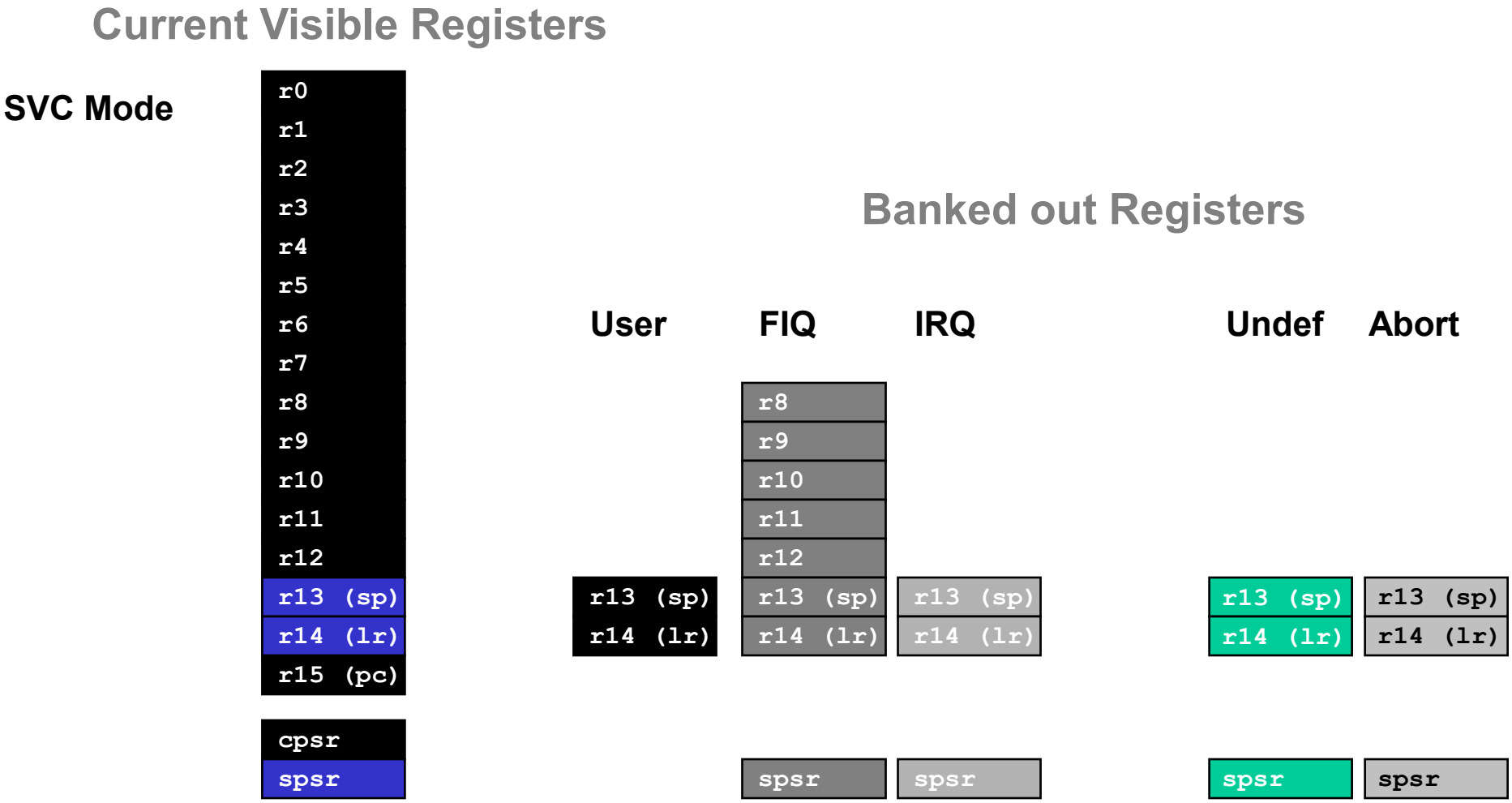
The ARM Register Set



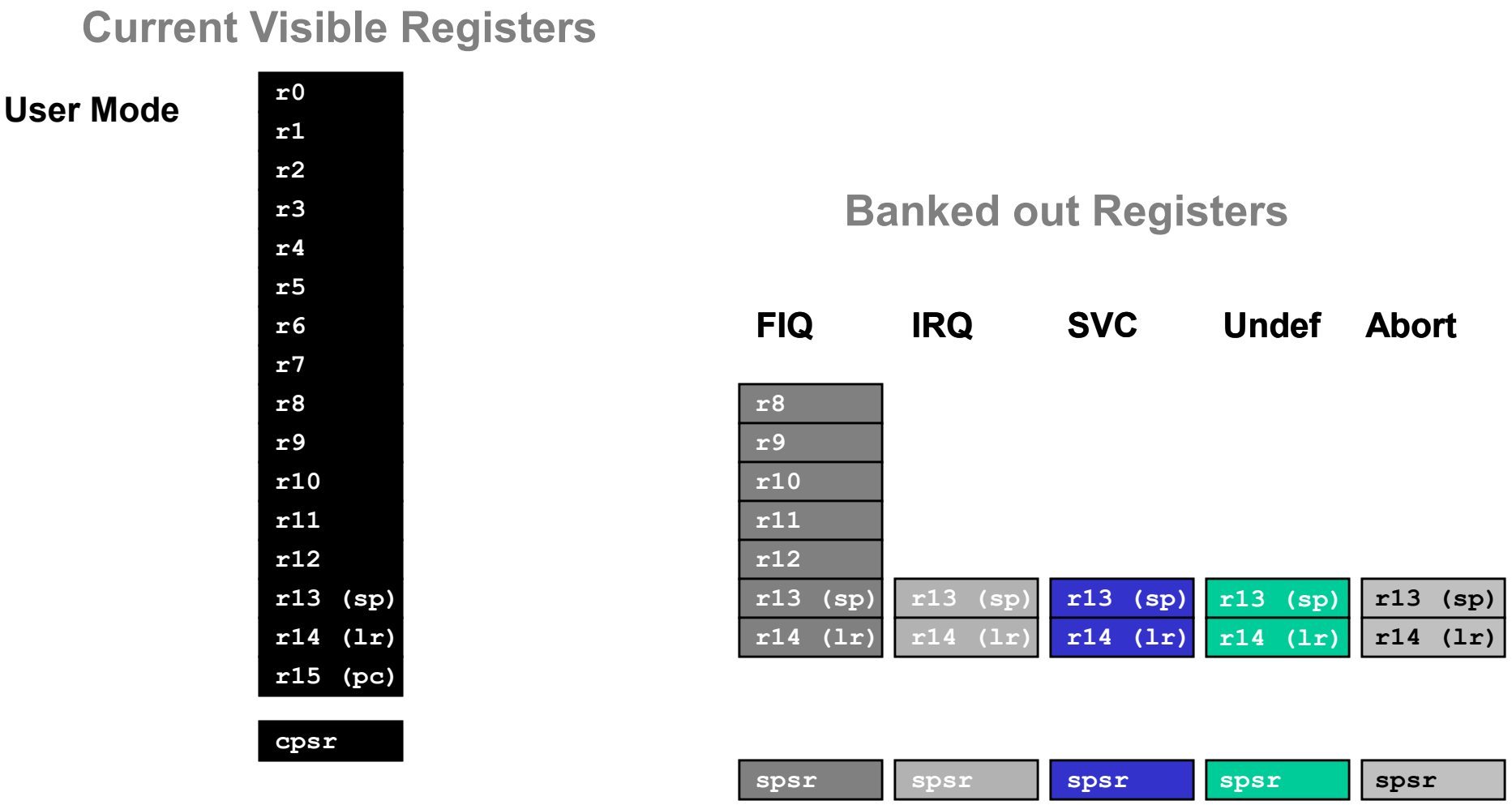
The ARM Register Set



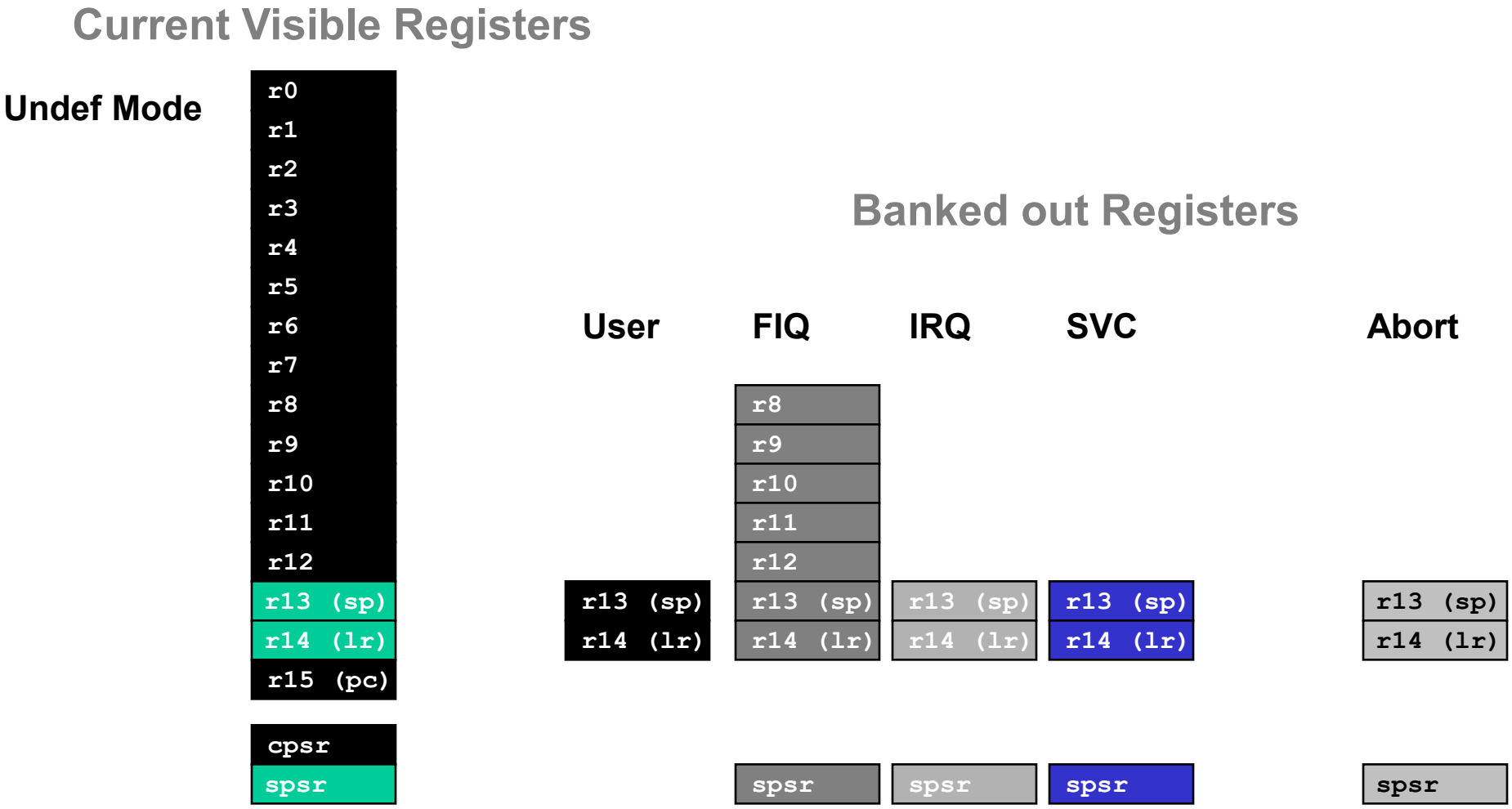
The ARM Register Set



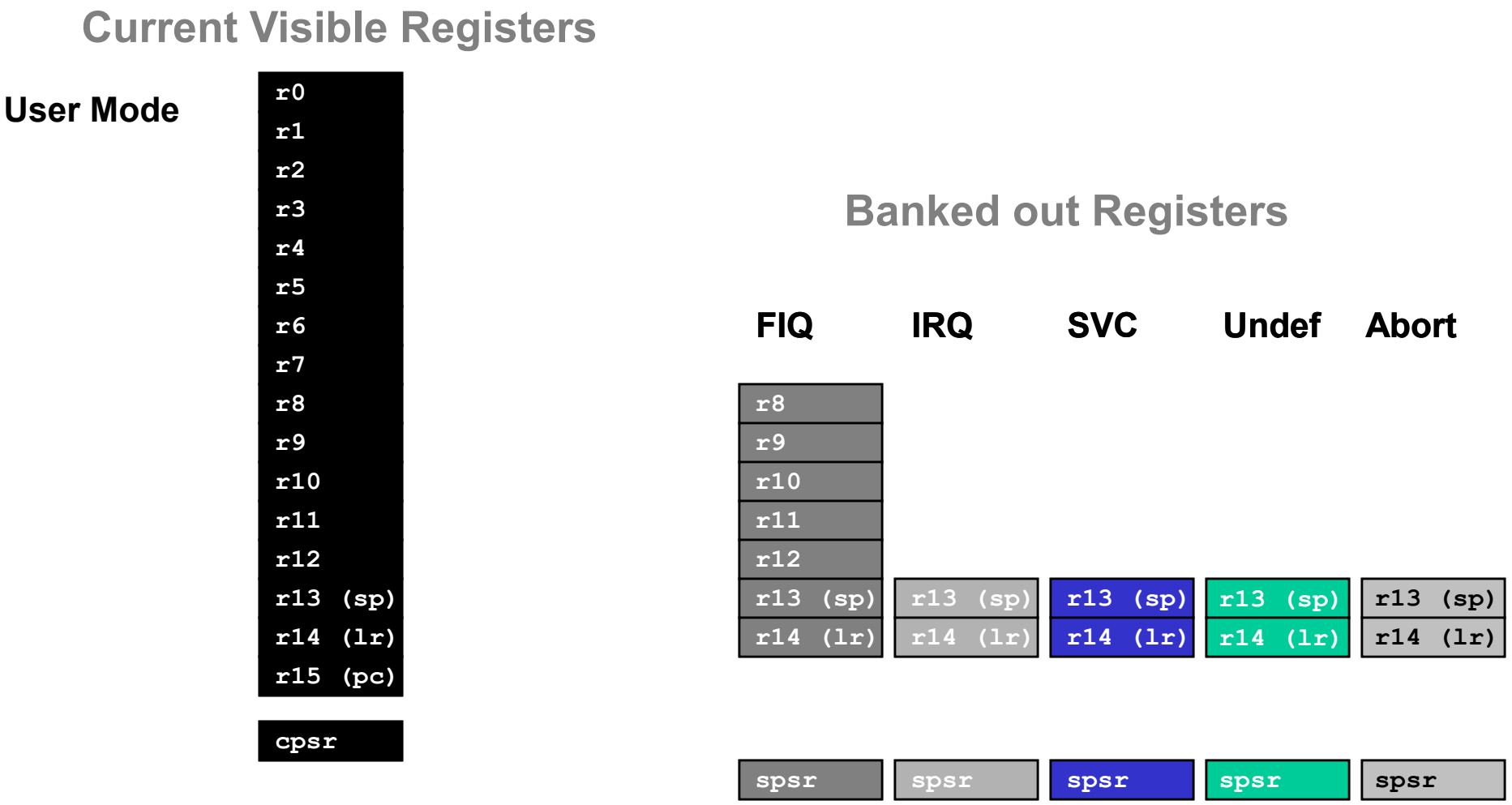
The ARM Register Set



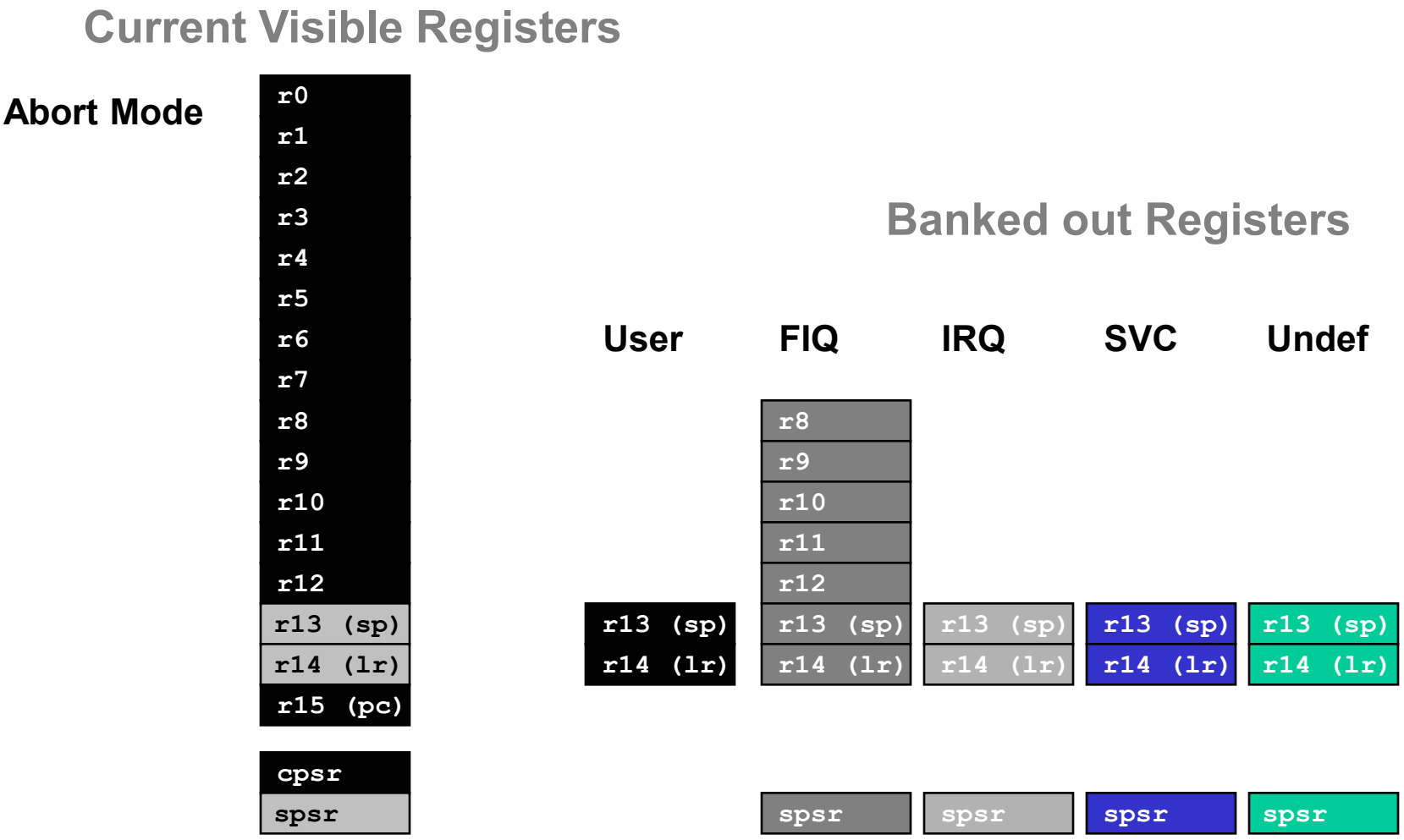
The ARM Register Set



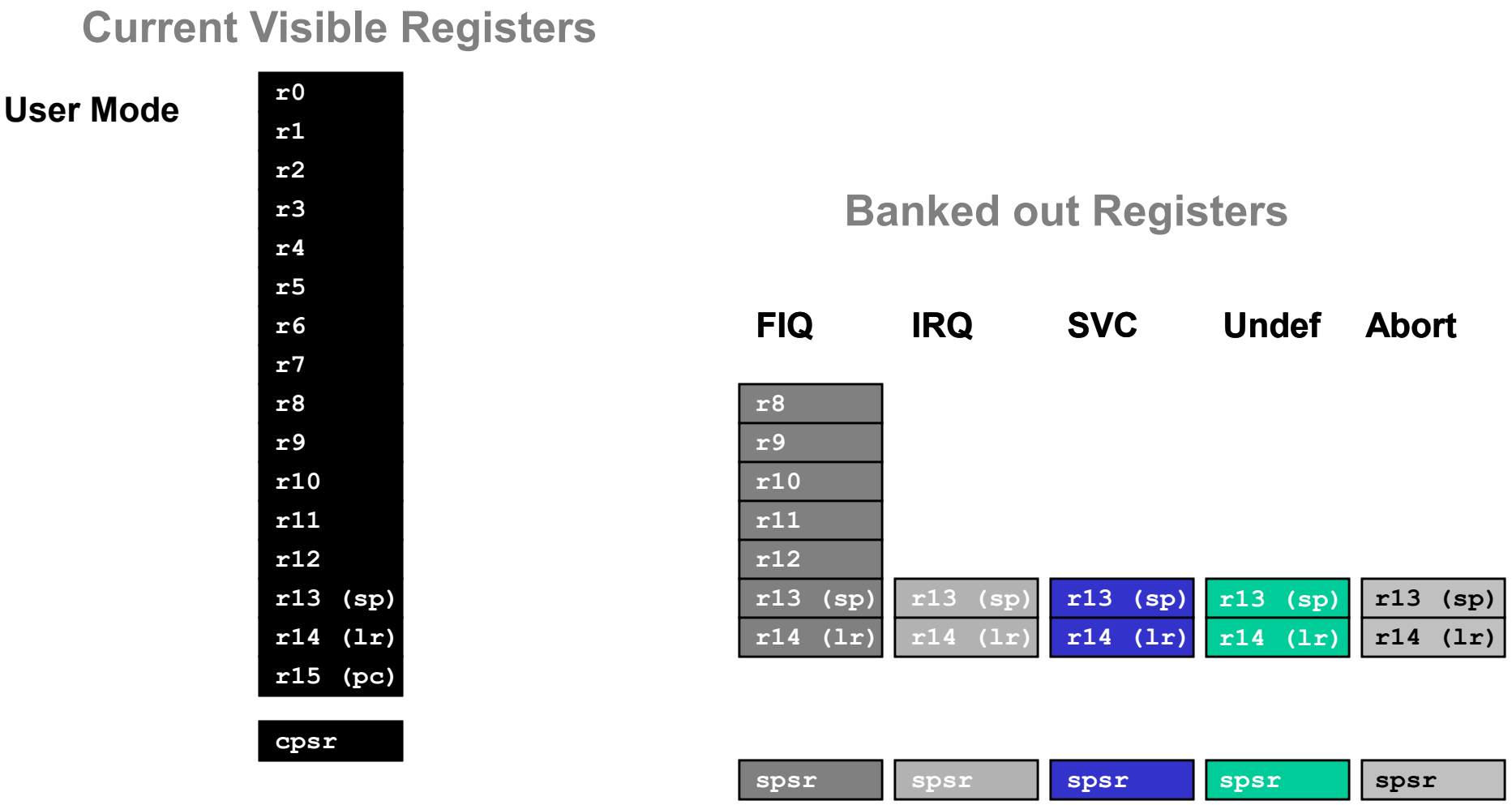
The ARM Register Set



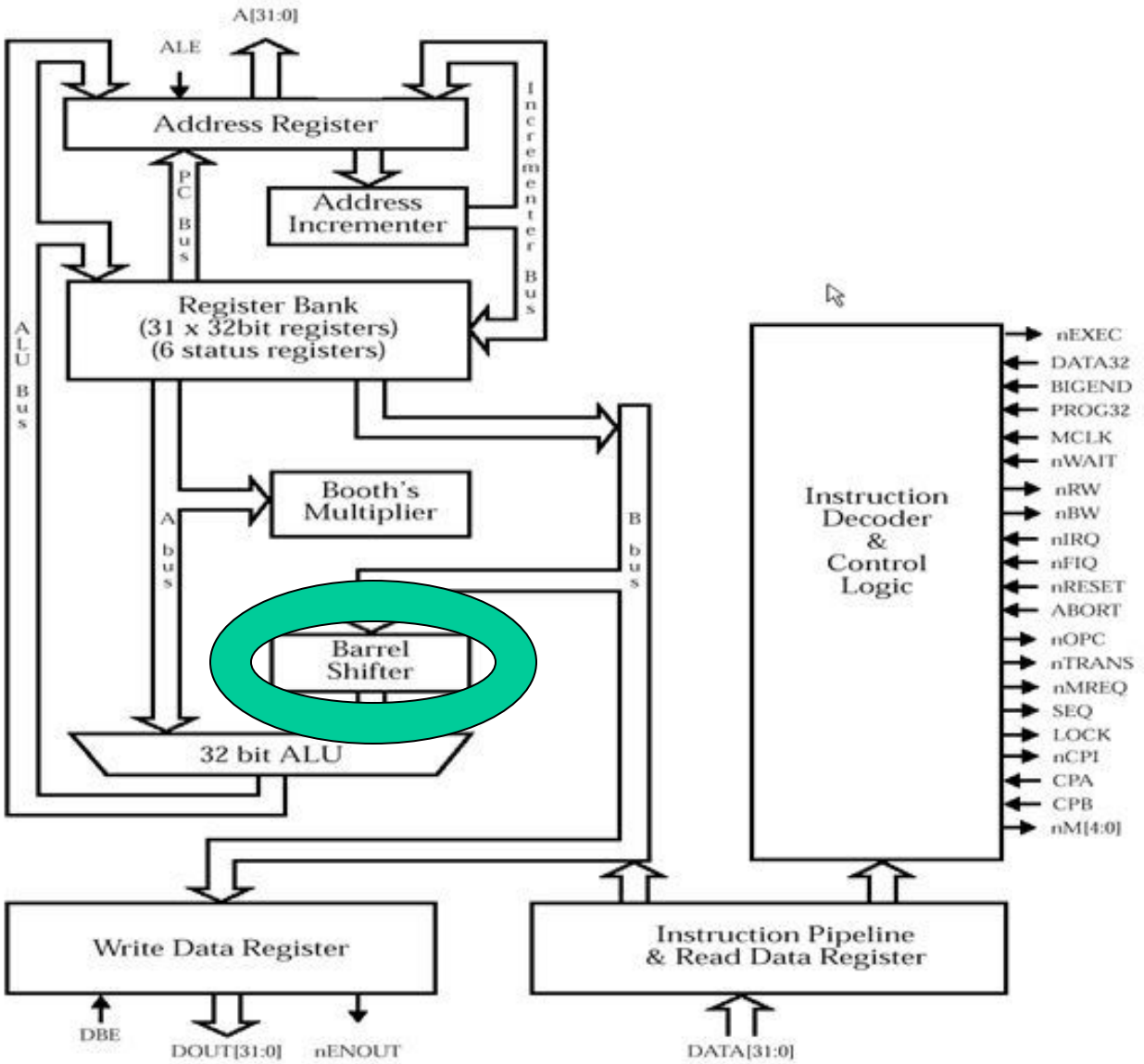
The ARM Register Set



The ARM Register Set

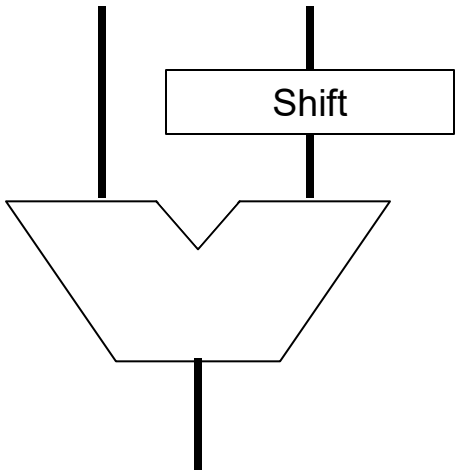


ARMv7 Data Path



Other “goodies”

The ALU in ARM has a “free shift” in front of it:



Logical shift left (lsl)
Logical shift right (lsr)
Arithmetic shift right (asr)
Rotate right (ror)

Conditional execution

In LC-3, the BR instruction uses the condition code register NZP

In ARM, the condition code register is called the CPSR (Current Program Status Register). It has:

N - Negative Result from ALU

Z - Zero result from ALU

C - ALU operation had a carry-out of MSbit

V - ALU operation had a twos-complement overflow

(plus bits to handle interrupts, etc.)

In ARM, *ALL* instructions can "test" these bits

Branching…

B label or BL label (branch and link)

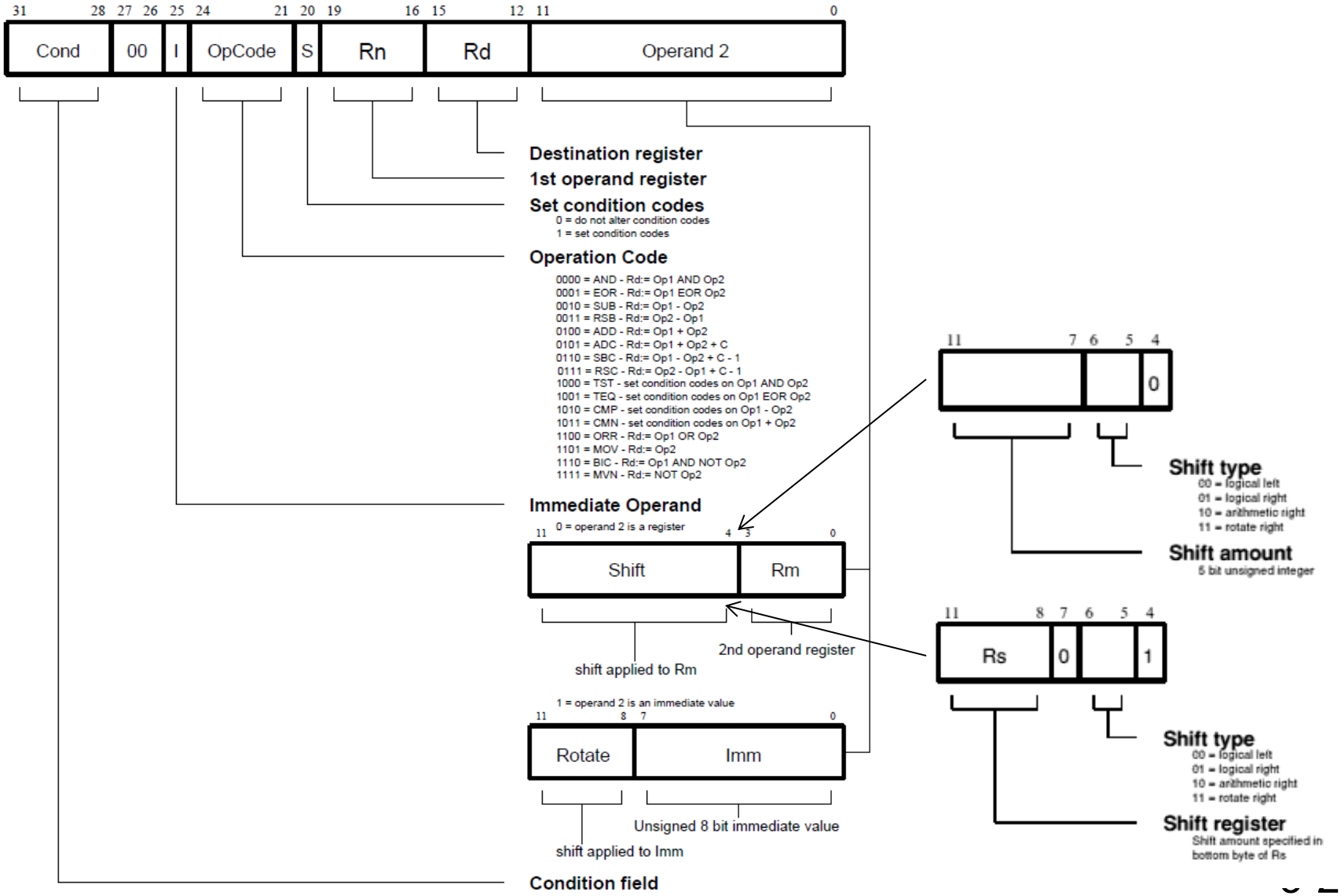
Conditional branching B*suffix* label…

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero

more…

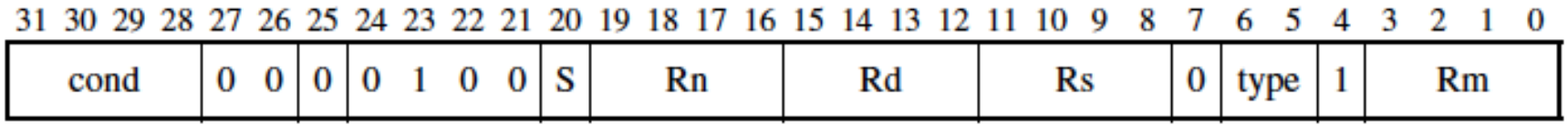
Suffix	Flags	Meaning
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

ARM data processing instruction format



Putting it all together: the ARM ADD instruction

```
ADD{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Ignore "cond" for now

type = lsl, lsr, asr, or ror and applies to Rm

Examples:

ADD R1,R2,R3	**r1 = r2+r3**
ADDS R1,R2,R3	**r1 = r2+r3 and the flags are set**
ADDS R1,R2,R3 lsl R4	**r1 = r2+(r3 lsl r4)**

Arithmetic opcodes that work like ADD

op r1, r2, r3

SUB	**r1 = r2-r3**	**Subtract**
RSB	**r1 = r3-r2**	**Reverse subtract**
AND	**r1 = r1 & r2**	**And**
ORR	**r1 = r1 | r2**	**Or**
EOR	**r1 = r1 ^ r2**	**Exclusive or**

not a comprehensive list!

ARM has about 200 instructions depending on how you count!

Some two-operand opcodes

***op* r1, r2**

CMP set flags based on r1-r2
MOV r1, r2

note can use an immediate if needed…

```
CMP r1,r2
MOV r1,#0
```

and shift…

```
MOV r1,r2, lsl #2
```

Loads and stores

LDR*type* R1, [R2, #imm]

STR*type* R1, [R2, #imm]

What's *type*? Something LC-3 didn't have:

type is one of

- **B for byte (8b)**
- **H for halfword (16b)**
- **(blank) for word (32b)**
- **D for doubleword (64b)**

Also can add an "S" for *sign extension*

so you can have

LDRSB R1, [R2,#4] **; Loads a byte into a 32-bit register and sign extends it**

All kinds of ways to get to memory…

[Rx, #imm]
[Rx, +Ry] [Rx, -Ry]
[Rx, ±Ry, *shiftop* #imm]

also add ! to update the Ry register afterwards
LDRD R1,[R2, -R3, lsl #3]!

effective address is R2-(R3<<3)
after the load executes: R3 = R2-(R3<<3)

or this syntax to update the register before the load
LDRD R1,R2,[-R3] lsl #3
R3 = R2-R3<<3 *before* the load executes
then the effective address is R3

How do you get an address in a register?

Same conundrum as in LC-3…

In LC-3

LEA R1,*label*

In ARM

LDR R1,=*label* puts the value of "label" in R1

Calls and returns

A call is a BL *label* for “branch and link”

LR (R14) gets the current PC+4

A return is via a BX *Rx, or usually* BX LR

Quick example

Count the number of positive values of an array of ints, A[ ],there are 16 values to investigate…

```
            LDR R0,=A                   ;Address of A in R0
            MOV R1,#15                  ;Countdown value
            MOV R2,#0                   ;Initialize counter
Label:      LDR R3,[R0,+R1 lsl #2]      ;Get an array val
            CMP R3,#0                   ;Test it
            BLT Noadd                   ;Ignore neg vals
            ADD R2,R2,#1                ;Increment count
Noadd:      SUBS R1,R1,#1               ;Dec counter
            BGE Label                   ;More if not done
```

Ok some weird (cool) stuff

ARM has conditional execution for (nearly) all instructions.

Those conditions (LT, LE, GT, …) can be appended to an opcode that allows it

So *instead of this*

```
            CMP R3,#0
            BLT Noadd
            ADD R2,R2,#1
Noadd:      …
```

we could have done this

```
            CMP R3,#0
            ADDGE R2,R2,#1
```

Quick example

Count the number of positive values of an array, there are 16 values to investigate…

```
                LDR R0,=A
                MOV R1,#15
                MOV R2,#0
Label:          LDR R3,[R0,+R1 lsl #2]
                CMP R3,#0
                ADDGE R2,R2,#1              ;Increment if >= 0
                SUBS R1,R1,#1
                BGE Label
```

Stacks!

Often load or store multiple…

STMxx sp!, {register list} **reg order doesn't matter here**
! means post-update SP, xx for + or -

LDMxx sp!, {register list}

where xx is the kind of stack
FD - full (sp points to top of stack), descending (--sp to push)**
FA - full, ascending (sp++ to push)
also
ED – empty descending
EA – empty ascending
**** what the ARM calling convention uses, same as LC-3**

Unix calling convention for ARM

r0-r3 are the argument and scratch registers;

r0-r1 are also the return value registers

r4-r8, r10-r11 are callee-save registers

r9 special use

r11 is the frame pointer

r12 is a temporary (caller-save)

r13 the link register

r14 the stack pointer

r15 the PC

A cross compiler you can get

For Mac OSX:

http://sourceforge.net/projects/yagarto/

free, pre-built binaries. Be sure to read the "read me first" before you install

then try

eabi-none-arm-gcc -S prog.c

makes a prog.s assembly file

Using GCC cross compiler…

```
int a[16];
main() {
int i, j;

        for (i = 15; i >= 0; i--)
          if (a[i] >= 0) j++;

        printf("%d\n", j);
}
```

```
.LC0: ascii  "%d\012\000"
      .text
main:  stmfd   sp!, {fp, lr}      ; save fp and lr (store multiple)
       add     fp, sp, #4         ; set up frame pointer ☺
       sub     sp, sp, #8         ; int i, j          //make space for two local vars
       mov     r3, #15            ; for (i = 15; …
       str     r3, [fp, #-8]      ;
       b       .L2                ;                   //go to test part of for loop
.L4:   ldr     r3, .L5            ; if (a[i] >= 0)  //load in base of 'a'
       ldr     r2, [fp, #-8]      ;                   //get I
       ldr     r3, [r3, r2, lsl #2]  ;                //get a[i]
       cmp     r3, #0             ;
       blt     .L3                ;                   //branch less than over if part
       ldr     r3, [fp, #-12]     ; j++               //if part: get j
       add     r3, r3, #1         ;                   //increment j
       str     r3, [fp, #-12]     ;                   //update j.
.L3:   ldr     r3, [fp, #-8]      ; …i--)             //reinitialize part of for
       sub     r3, r3, #1         ;                   //decrement i
       str     r3, [fp, #-8]      ;                   //update i
.L2:   ldr     r3, [fp, #-8]      ; …i >= 0;…    // test part of for loop
       cmp     r3, #0
       bge     .L4                ;                   //back to loop body if i >= 0
       ldr     r0, .L6            ; printf("%d\n, j)//get the address of the format string for printf
       ldr     r1, [fp, #-12]     ;                   //get the value of j
       bl      printf             ;                   //branch and link (call) printf
       sub     sp, fp, #4         ; pop the stack frame
       ldmfd   sp!, {fp, lr}      ; restore registers
       bx      lr                 ; return
.L5:   .word   a                  ; pointer to a[0]
.L6    .word   .LC0               ; pointer to literal "%d\n"
, ,
```

Thumb

ROM is precious on an embedded device

Can “save space” in ROM using *Thumb*

Thumb is a proper subset of ARM ISA

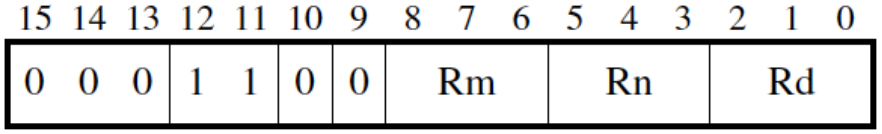
Instructions are 16b wide, which reduces what you can do.

Here’s a Thumb ADD encoding:

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

`ADDS <Rd>,<Rn>,<Rm>` Outside IT block.

`ADD<c> <Rd>,<Rn>,<Rm>` Inside IT block.

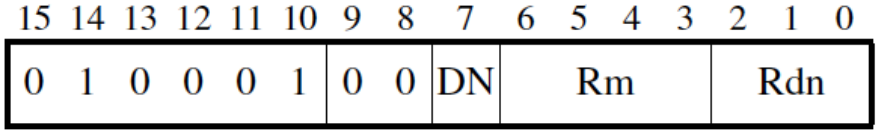



```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

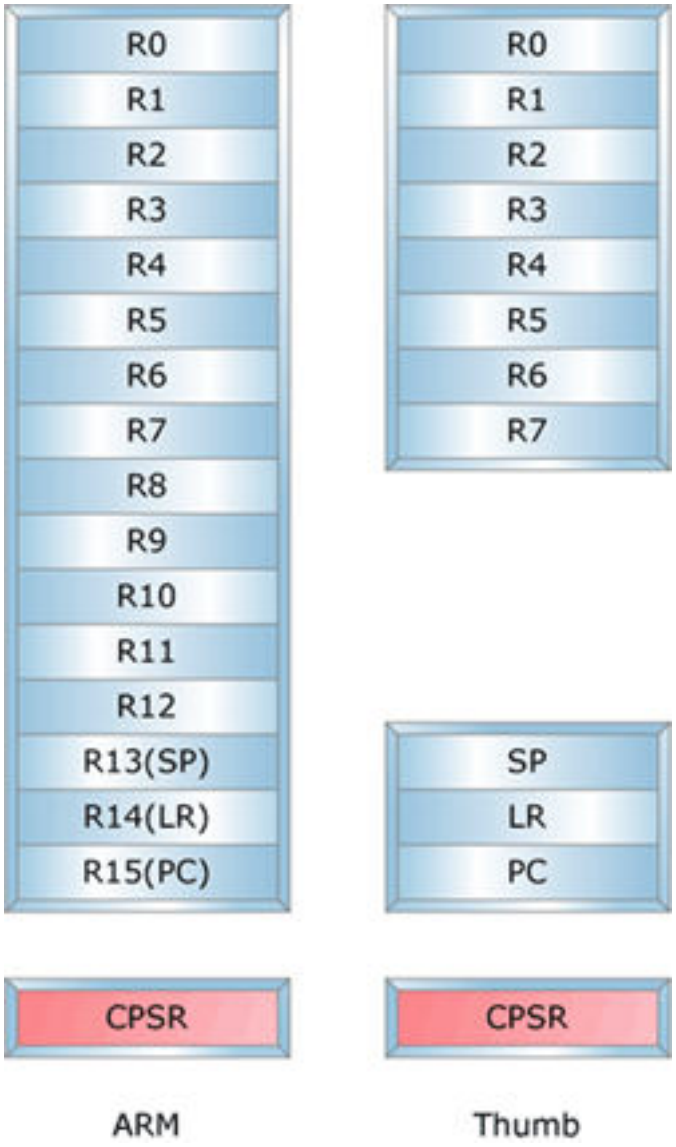
Encoding T2 ARMv6T2, ARMv7 if <Rdn> and <Rm> are both from R0-R7

ARMv4T, ARMv5T*, ARMv6*, ARMv7 otherwise

`ADD<c> <Rdn>,<Rm>` If <Rdn> is the PC, must be outside or last in IT block.




```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```


R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13(SP)
R14(LR)
R15(PC)
CPSR
ARM
R0
R1
R2
R3
R4
R5
R6
R7
SP
LR
PC
CPSR
Thumb

Has Push, Pop

ARM

```
main:  stmfd   sp!, {fp, lr}
       add     fp, sp, #4
       sub     sp, sp, #8
       mov     r3, #15
       str     r3, [fp, #-8]
       b       .L2
.L4:   ldr     r3, .L5
       ldr     r2, [fp, #-8]
       ldr     r3, [r3, r2, lsl #2]
       cmp     r3, #0
       blt     .L3
       ldr     r3, [fp, #-12]
       add     r3, r3, #1
       str     r3, [fp, #-12]
.L3:   ldr     r3, [fp, #-8]
       sub     r3, r3, #1
       str     r3, [fp, #-8]
.L2:   ldr     r3, [fp, #-8]
       cmp     r3, #0
       bge     .L4
       ldr     r0, .L6
       ldr     r1, [fp, #-12]
       bl      printf
       mov     r0, r3
       sub     sp, fp, #4
       ldmfd   sp!, {fp, lr}
       bx      lr
```

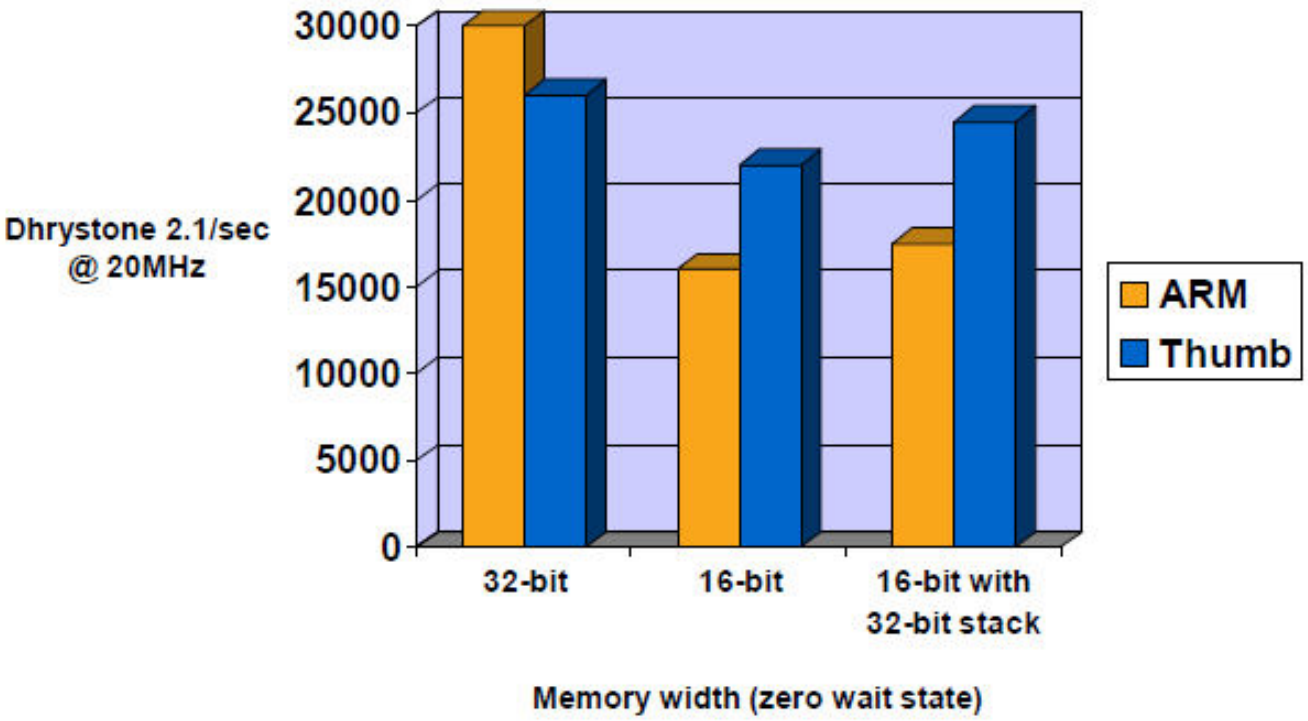
27 instructions x 4B each
total =108 Bytes

Thumb

```
main:       push    {r7, lr}
       sub     sp, sp, #8
       add     r7, sp, #0
       mov     r3, #15
       str     r3, [r7, #4]
       b       .L2
.L4:        ldr     r3, .L5
       ldr     r2, [r7, #4]
       lsl     r2, r2, #2
       ldr     r3, [r2, r3]
       cmp     r3, #0
       blt     .L3
       ldr     r3, [r7]
       add     r3, r3, #1
       str     r3, [r7]
.L3:        ldr     r3, [r7, #4]
       sub     r3, r3, #1
       str     r3, [r7, #4]
.L2:        ldr     r3, [r7, #4]
       cmp     r3, #0
       bge     .L4
       ldr     r2, .L6
       ldr     r3, [r7]
       mov     r0, r2
       mov     r1, r3
       bl      printf
       mov     r0, r3
       mov     sp, r7
       add     sp, sp, #8
       pop     {r7}
       pop     {r1}
       bx      r1
```

32 instructions x 2B each
total = 64 Bytes

ARM and Thumb Performance



Floating Point (!!!)

Uses a separate register file

Has either 32 32b registers called S0 to S31

or 16 64b registers called D0 to D15

To get values from "regular" registers in/out of these, you can use VMOV

VMOV.*size* S3, R1

where *size* is .8 or .16 or .32

VADD.*F32 Sd, Sn, Sm* *also VMUL, VDIV, VSUB, VSQRT…*

VADD.*F64 Dd, Dn, Dm*

Smart C Programming

First 4 parameters passed in registers

In loops always make the loop variable count down, if possible

Size of local variables

```
int wordsize(int a)
{
  a=a+1;
  return a;
}
int halfsize(short b)
{
  b=b+1;
  return b;
}

int bytesize(char c)
{
  c=c+1;
  return c;
}
```

```
wordsize
ADD r0,r0,#1
BX lr

halfsize
ADD r0,r0,#1
MOV r0,r0,LSL #16
MOV r0,r0,ASR #16
BX lr

bytesize
ADD r0,r0,#1
AND r0,r0,#0xFF
BX lr
```

Resources

http://www.arm.com/support/university/