C++ Real-Time Audio Programming with Bela

Dr Andrew McPherson

Centre for Digital Music
School of Electronic Engineering and Computer Science
Queen Mary University of London

Founder and Director, Bela

Course topics

Programming topics

Working in real time
Buffers and arrays
Parameter control
Classes and objects
Analog and digital I/O
Filtering
Timing in real time
Circular buffers
State machines
MIDI
Block-based processing
Threads
Fixed point arithmetic
ARM assembly language

Today

Music/audio topics

Oscillators
Samples
Wavetables
Control voltages
Gates and triggers
Filters
Metronomes and clocks
Delays and delay-based effects
Envelopes
ADSR
MIDI
Additive synthesis
Phase vocoders
Impulse reverb

Lecture 21: ARM assembly language

What you'll learn today:

Basics of assembly language instructions on ARM
Calling assembly language code from C++
Floating point in assembly language

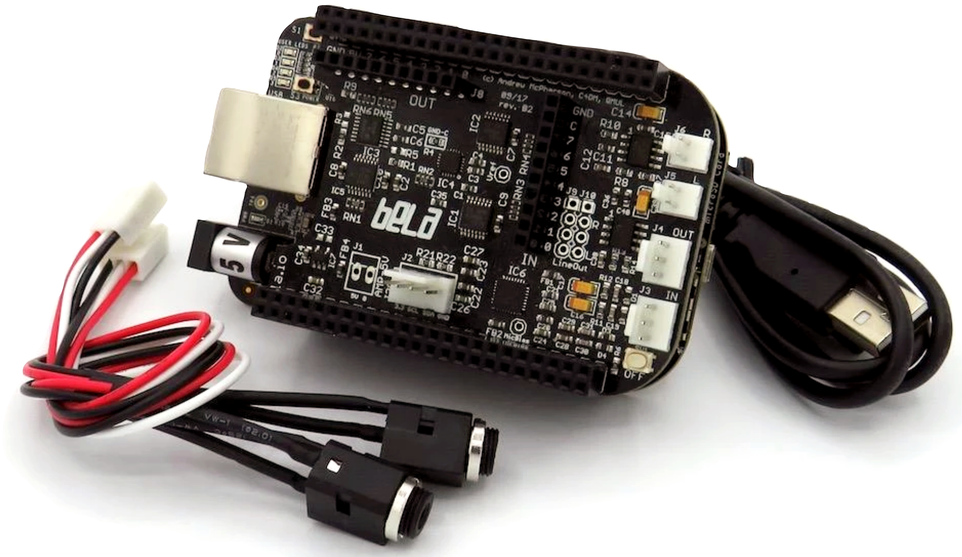
What you'll make today:

Example functions written in assembly

Companion materials:

github.com/BelaPlatform/bela-online-course

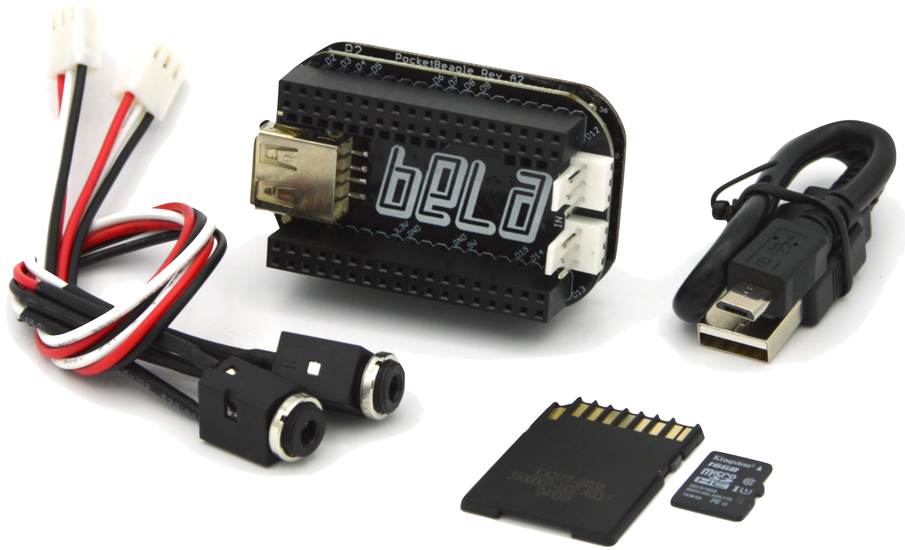
What you'll need



Bela Starter Kit

or

[shop.bela.io]



Bela Mini Starter Kit

Ways of coding

Machine Code (binary)

```
0000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0000010 02 00 28 00 01 00 00 00 15 83 00 00 34 00 00 00
0000020 20 92 00 00 02 00 00 05 34 00 20 00 09 00 28 00
0000030 26 00 23 00 01 00 00 70 44 04 00 00 44 84 00 00
0000040 44 84 00 00 08 00 00 00 08 00 00 00 04 00 00 00
0000050 04 00 00 00 06 00 00 00 34 00 00 00 34 80 00 00
0000060 34 80 00 00 20 01 00 00 20 01 00 00 05 00 00 00
0000070 04 00 00 00 03 00 00 00 54 01 00 00 54 81 00 00
0000080 54 81 00 00 13 00 00 00 13 00 00 00 04 00 00 00
0000090 01 00 00 00 01 00 00 00 00 00 00 00 00 00 80 00 00
00000a0 00 80 00 00 50 04 00 00 50 04 00 00 05 00 00 00
```

C / C++

```
int main(int argc, char *argv[])
{
   float x = 0, y = 1.0;

   for(i = 0; i < 10; i++)
   {
      float z = test(x, y);
      printf("%f\n", z);
      x += 0.1;
      y -= 0.1;
   }

   return 0;
}
```

Assembly Language

```
ldr   r0, [r7, #8]
bl factorial
str   r0, [r7, #12]
.loc 1 39 0
movw  r3, #:lower16:.LC0
movt  r3, #:upper16:.LC0
mov   r0, r3
ldr   r1, [r7, #8]
ldr   r2, [r7, #12]
bl printf
```

Intrepreted and Bytecode Languages (Python, Javascript, PHP, ...)

```
def main():
    parser = argparse.ArgumentParser(usage=__doc__)
    parser.add_argument("--order", type=int, \
      default=3, help="order of Bessel function")
    parser.add_argument("--output", \
      default="plot.png", help="output image file")
    args = parser.parse_args()

    f = lambda x: -special.jv(args.order, x)
    sol = optimize.minimize(f, 1.0)
```

Assembly language

- Machine code is the binary data that actually runs on the processor
- Assembly language is the human readable version of machine code
  - 1 line of assembly → 1 machine instruction
    - With a few exceptions (labels, etc.) which we'll see later
  - Of all code we can write, it's closest to what the hardware actually does
  - Compare this to higher level languages, where there is no straightforward relationship between the text of the code and the number of resulting instructions
- So what does a CPU actually do?
  - Arithmetic and logical operations
    - Add, subtract, multiply, bit shift, bitwise AND, bitwise OR, etc.
  - Load and store values from memory (RAM)
  - Testing and branching on various conditions
    - e.g. "branch if result is zero"

C vs. assembly

- C and C++ compile to assembly language
  - ...and then build to machine code which the processor executes
- Why write code in C versus assembly?

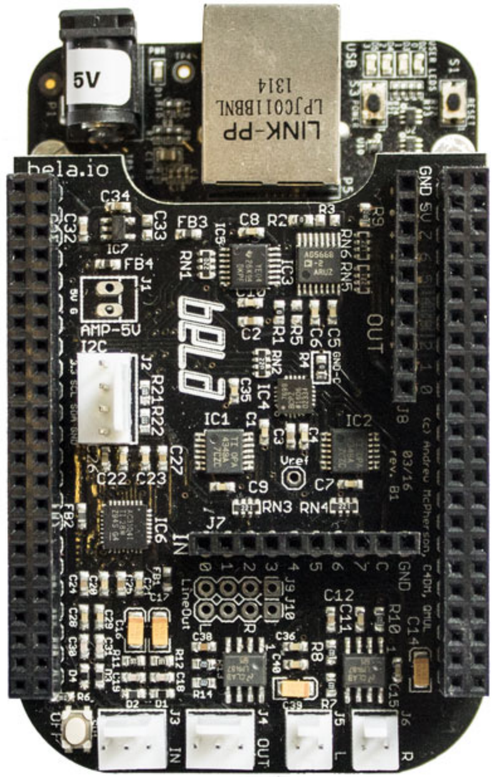
Assembly Language

- Closest to the hardware
  - Know exactly what the CPU does and how long it will take
  - Possibly very efficient
- CPU-specific
  - Every CPU architecture has its own, distinct assembly language
- Only primitive instructions
  - Programmer must make their own variables, functions, loops, conditionals, etc.

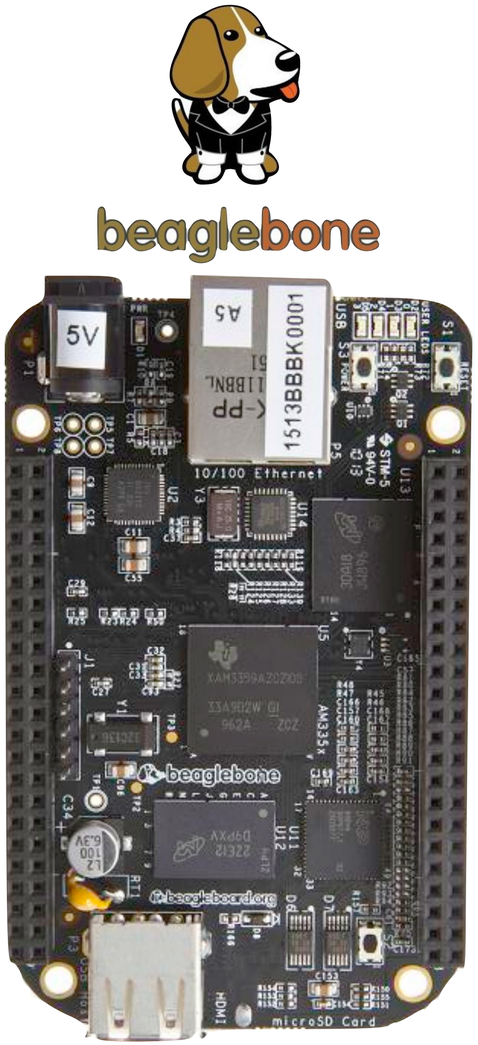
C

- Procedural language
  - Similar to how hardware works, but no direct analogy to CPU instructions
- CPU-independent
  - Assuming use of only standard libraries
- Higher-level instructions
  - Variables, functions, conditionals, `for()` and `while()` loops, etc.
- One line of C might produce many lines of assembly

The Bela CPU



uses



powered by

Texas Instruments AM3358 “Sitara” system-on-chip

CPU core:

ARM Cortex-A8 (1GHz)

ARM architecture

- Many different types of ARM processor
  - Most are 32-bit processors; some more recent models are 64-bit
- ARM microcontrollers:
  - ARM Cortex-M0; smallest / cheapest microcontroller
  - ARM Cortex-M3; more sophisticated
  - ARM Cortex-M4; M3 plus DSP instructions, optional FPU
- ARM application processors:
  - ARM11 (older generation) used by original Raspberry Pi
  - ARM Cortex-A8; up to 1GHz, vector floating-point; SIMD instructions
  - ARM Cortex-A7, Cortex-A9, Cortex-A15, etc.; more advanced than A8, multicore support
  - ARM Cortex-A53, Cortex-A57, Cortex-A72, etc.; newest 64-bit processors
- Similar instruction set across all processors
  - Some processors have instructions not found on others, but also the implementation changes
- BeagleBone Black uses an ARM Cortex-A8
  - ARMv7-A instruction set (see companion materials for reference)

ARM instructions

Rd, Rn, etc. What are these?

Registers

Operation		§	Assembler	S updates	Action	Notes
Add	Add		`ADD{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2	N
	with carry		`ADC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2 + Carry	N
	wide	T2	`ADD Rd, Rn, #<imm12>`		Rd := Rn + imm12, imm12 range 0-4095	T, P
	saturating {doubled}	5E	`Q{D}ADD Rd, Rm, Rn`		Rd := SAT(Rm + Rn) doubled: Rd := SAT(Rm + SAT(Rn * 2))	Q
Address	Form PC-relative address		`ADR Rd, <label>`		Rd := <label>, for <label> range from current instruction see Note L	N, L
Subtract	Subtract		`SUB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2	N
	with carry		`SBC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2 – NOT(Carry)	N
	wide	T2	`SUB Rd, Rn, #<imm12>`		Rd := Rn – imm12, imm12 range 0-4095	T, P
	reverse subtract		`RSB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn	N
	reverse subtract with carry		`RSC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn – NOT(Carry)	A
	saturating {doubled}	5E	`Q{D}SUB Rd, Rm, Rn`		Rd := SAT(Rm – Rn) doubled: Rd := SAT(Rm – SAT(Rn * 2))	Q
	Exception return without stack		`SUBS PC, LR, #<imm8>`	N Z C V	PC = LR – imm8, CPSR = SPSR(current mode), imm8 range 0-255.	
Parallel arithmetic	Halfword-wise addition	6	`<prefix>ADD16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[31:16], Rd[15:0] := Rn[15:0] + Rm[15:0]	G
	Halfword-wise subtraction	6	`<prefix>SUB16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[31:16], Rd[15:0] := Rn[15:0] – Rm[15:0]	G
	Byte-wise addition	6	`<prefix>ADD8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] + Rm[31:24], Rd[23:16] := Rn[23:16] + Rm[23:16], Rd[15:8] := Rn[15:8] + Rm[15:8], Rd[7:0] := Rn[7:0] + Rm[7:0]	G
	Byte-wise subtraction	6	`<prefix>SUB8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] – Rm[31:24], Rd[23:16] := Rn[23:16] – Rm[23:16], Rd[15:8] := Rn[15:8] – Rm[15:8], Rd[7:0] := Rn[7:0] – Rm[7:0]	G
	Halfword-wise exchange, add, subtract	6	`<prefix>ASX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[15:0], Rd[15:0] := Rn[15:0] – Rm[31:16]	G
	Halfword-wise exchange, subtract, add	6	`<prefix>SAX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[15:0], Rd[15:0] := Rn[15:0] + Rm[31:16]	G
	Unsigned sum of absolute differences	6	`USAD8 Rd, Rm, Rs`		Rd := Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
	and accumulate	6	`USADA8 Rd, Rm, Rs, Rn`		Rd := Rn + Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
Saturate	Signed saturate word, right shift	6	`SSAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := SignedSat((Rm ASR sh), sat). `<sat>` range 1-32, `<sh>` range 1-31.	Q, R
	Signed saturate word, left shift	6	`SSAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := SignedSat((Rm LSL sh), sat). `<sat>` range 1-32, `<sh>` range 0-31.	Q
	Signed saturate two halfwords	6	`SSAT16 Rd, #<sat>, Rm`		Rd[31:16] := SignedSat(Rm[31:16], sat), Rd[15:0] := SignedSat(Rm[15:0], sat). `<sat>` range 1-16.	Q
	Unsigned saturate word, right shift	6	`USAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := UnsignedSat((Rm ASR sh), sat). `<sat>` range 0-31, `<sh>` range 1-31.	Q, R
	Unsigned saturate word, left shift	6	`USAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := UnsignedSat((Rm LSL sh), sat). `<sat>` range 0-31, `<sh>` range 0-31.	Q
	Unsigned saturate two halfwords	6	`USAT16 Rd, #<sat>, Rm`		Rd[31:16] := UnsignedSat(Rm[31:16], sat), Rd[15:0] := UnsignedSat(Rm[15:0], sat). `<sat>` range 0-15.	Q

Name and arguments of instruction

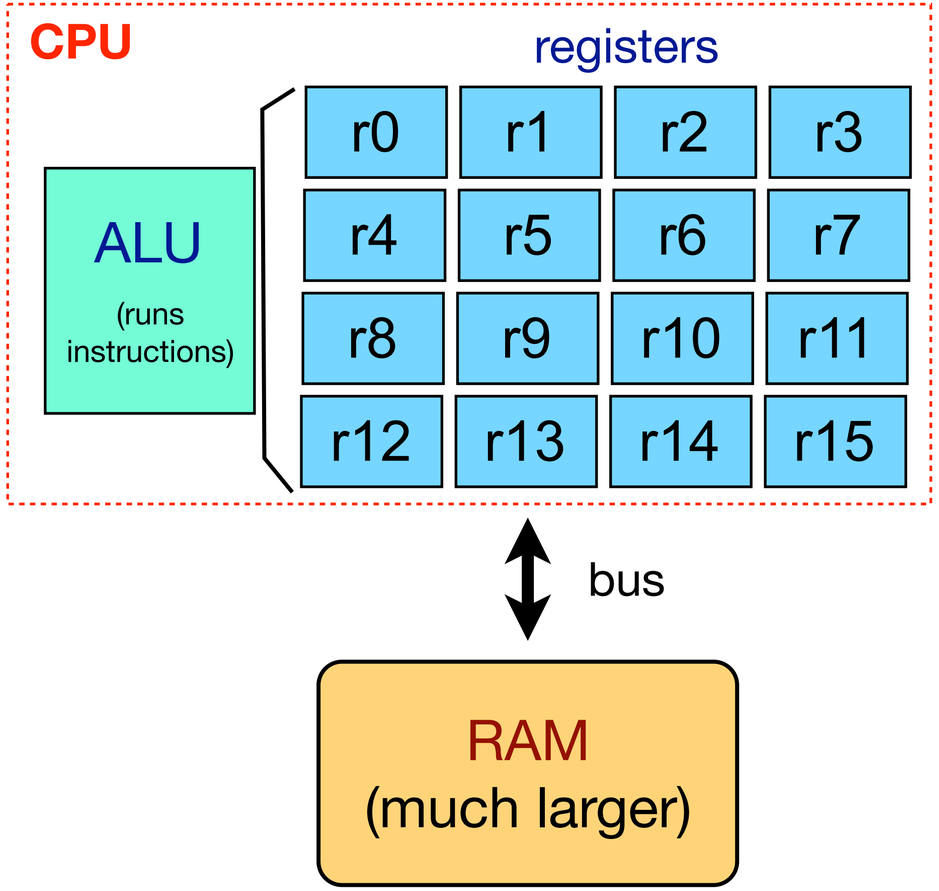
What the instruction does

Variables

- In C and C++, we work with variables
  - We have seen global and local variables
  - Variables are slots in memory (RAM) which hold values
  - Type (`int`, `float`, etc.) tells us two things:
    - How big (how many bytes) the slot in memory needs to be
    - How we should interpret the data in the slot
- In C/C++, a pointer tells us where in memory a given variable is located
  - Use the `&` sign to get the pointer to a variable
    - i.e. `&x` is where the variable `x` lives in memory
  - Use the `*` sign to dereference a pointer
    - i.e. `*p` tells us what value is located at memory address `p`
  - The `[]` postscript for arrays works similarly to `*`
    - i.e. `a[4]` tells us what is in the memory slot 4 elements into `a`

Variables and registers

- In assembly, we don't have variables
  - At least not in a simple form we can use directly in single instructions
- Instead all our operations work on registers
  - Registers are slots within the CPU itself that hold numerical values
  - ARM processors have 16 registers
  - Other CPU types differ in their register structure
  - For all arithmetic instructions and most others, both input and output are registers
    - Just a few exceptions to be detailed later



Registers

- What if we need more than 16 values?
  - Move values between registers and memory
  - Load instructions move value from memory to register
  - Store instructions move from register to memory
- Recall that C/C++ variables are typically stored in RAM
  - Thus, using a variable might involve three instructions:
    - Load from memory into register
    - The arithmetic / logic operation itself
    - Store from register into memory (if the variable changes)
  - This happens unseen to the C programmer
  - Caveat: the compiler might optimise the code so that a variable only ever uses a register
- Load and store instructions often incur a delay
  - Could be at least several CPU cycles, because RAM is slower than register access
  - If we're not careful, RAM accesses can even become the bulk of the program execution time

ARM assembly example

C

```
a = a + b;  // Add a and b
```

ARM Assembly

```
ldr r0, [r7, #0]   @ Load a from RAM into r0
ldr r1, [r7, #4]   @ Load b from RAM into r1
add r0, r0, r1     @ Add a and b
str r0, [r7, #0]   @ Store a back into RAM
```

- Instructions always lower case
- No semicolons needed
- Comments begin with `@` and continue to the end of line
- In this example, we assume `r7` holds the address of the variable `a` (`r7` ↔ `&a`) and that `b` is located in next memory address
  - The C compiler would keep track of where the variables are stored in memory to generate assembly code like this

ARM registers

- All ARM CPUs have 16 registers
  - Each register is 32 bits on Cortex-A8 and all Cortex-M processors
    - Registers are 64 bits on A53, A57 and other 64-bit processors
  - Therefore, one `int` fits in one register
    - Larger values like `long long` might require two registers
- Not all of the 16 registers are available for our code
  - Some of them have special functions by convention:
    - `r13` is `SP` (stack pointer); tells us where the stack ends
    - `r14` is `LR` (link register); tells us where to jump to at the end of the current function
    - `r15` is `PC` (program counter); tells CPU where next instruction comes from
    - Don't touch `r13` or `r15` under any circumstances!
    - Occasionally, `r7` and `r9` have special functions depending on system
  - We can use the rest of the registers as we like
    - Some must be restored when we're finished

ARM instructions

Operation		§	Assembler	S updates	Action	Notes
Add	Add		`ADD{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2	N
	with carry		`ADC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2 + Carry	N
	wide	T2	`ADD Rd, Rn, #<imm12>`		Rd := Rn + imm12, imm12 range 0-4095	T, P
	saturating {doubled}	5E	`Q{D}ADD Rd, Rm, Rn`		Rd := SAT(Rm + Rn) doubled: Rd := SAT(Rm + SAT(Rn * 2))	Q
Address	Form PC-relative address		`ADR Rd, <label>`		Rd := <label>, for <label> range from current instruction see Note L	N, L
Subtract	Subtract		`SUB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2	N
	with carry		`SBC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2 – NOT(Carry)	N
	wide	T2	`SUB Rd, Rn, #<imm12>`		Rd := Rn – imm12, imm12 range 0-4095	T, P
	reverse subtract		`RSB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn	N
	reverse subtract with carry		`RSC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn – NOT(Carry)	A
	saturating {doubled}	5E	`Q{D}SUB Rd, Rm, Rn`		Rd := SAT(Rm – Rn) doubled: Rd := SAT(Rm – SAT(Rn * 2))	Q
	Exception return without stack		`SUBS PC, LR, #<imm8>`	N Z C V	PC = LR – imm8, CPSR = SPSR(current mode), imm8 range 0-255.	
Parallel arithmetic	Halfword-wise addition	6	`<prefix>ADD16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[31:16], Rd[15:0] := Rn[15:0] + Rm[15:0]	G
	Halfword-wise subtraction	6	`<prefix>SUB16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[31:16], Rd[15:0] := Rn[15:0] – Rm[15:0]	G
	Byte-wise addition	6	`<prefix>ADD8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] + Rm[31:24], Rd[23:16] := Rn[23:16] + Rm[23:16], Rd[15:8] := Rn[15:8] + Rm[15:8], Rd[7:0] := Rn[7:0] + Rm[7:0]	G
	Byte-wise subtraction	6	`<prefix>SUB8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] – Rm[31:24], Rd[23:16] := Rn[23:16] – Rm[23:16], Rd[15:8] := Rn[15:8] – Rm[15:8], Rd[7:0] := Rn[7:0] – Rm[7:0]	G
	Halfword-wise exchange, add, subtract	6	`<prefix>ASX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[15:0], Rd[15:0] := Rn[15:0] – Rm[31:16]	G
	Halfword-wise exchange, subtract, add	6	`<prefix>SAX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[15:0], Rd[15:0] := Rn[15:0] + Rm[31:16]	G
	Unsigned sum of absolute differences	6	`USAD8 Rd, Rm, Rs`		Rd := Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
	and accumulate	6	`USADA8 Rd, Rm, Rs, Rn`		Rd := Rn + Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
Saturate	Signed saturate word, right shift	6	`SSAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := SignedSat((Rm ASR sh), sat). <sat> range 1-32, <sh> range 1-31.	Q, R
	Signed saturate word, left shift	6	`SSAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := SignedSat((Rm LSL sh), sat). <sat> range 1-32, <sh> range 0-31.	Q
	Signed saturate two halfwords	6	`SSAT16 Rd, #<sat>, Rm`		Rd[31:16] := SignedSat(Rm[31:16], sat), Rd[15:0] := SignedSat(Rm[15:0], sat). <sat> range 1-16.	Q
	Unsigned saturate word, right shift	6	`USAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := UnsignedSat((Rm ASR sh), sat). <sat> range 0-31, <sh> range 1-31.	Q, R
	Unsigned saturate word, left shift	6	`USAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := UnsignedSat((Rm LSL sh), sat). <sat> range 0-31, <sh> range 0-31.	Q
	Unsigned saturate two halfwords	6	`USAT16 Rd, #<sat>, Rm`		Rd[31:16] := UnsignedSat(Rm[31:16], sat), Rd[15:0] := UnsignedSat(Rm[15:0], sat). <sat> range 0-15.	Q

Form: outputs first, then inputs

What is “Operand2”?

<Operand2>

- ARM arithmetic/logical instructions are very flexible
- Second input argument can take several forms
  - Example: `sub Rd, Rn, <Operand2>`
    - Translates to: Rd := Rn - <Operand2>
- <Operand2> (or <Op2> for short) could be any one of:
  - An immediate value (a hard-coded number)
    - Restrictions on what value this can be; see ARM instruction reference for details
  - A register
    - any of `r0` to `r15`
  - A register bit-shifted by a constant
    - Equivalent to applying the C << or >> operators
  - A register bit-shifted by another register
    - ARM CPUs have a circuit called a barrel shifter which makes bit shifting like this extremely efficient

<Operand2>

<Op2> can be any of these things:

Notice: bit shifting doesn't take an extra instruction!

Flexible Operand 2	
Immediate value	`#<imm8m>`
Register, optionally shifted by constant (see below)	`Rm {, <opsh>}`
Register, logical shift left by register	`Rm, LSL Rs`
Register, logical shift right by register	`Rm, LSR Rs`
Register, arithmetic shift right by register	`Rm, ASR Rs`
Register, rotate right by register	`Rm, ROR Rs`

Register, optionally shifted by constant		
(No shift)	`Rm`	Same as `Rm, LSL #0`
Logical shift left	`Rm, LSL #<shift>`	Allowed shifts 0-31
Logical shift right	`Rm, LSR #<shift>`	Allowed shifts 1-32
Arithmetic shift right	`Rm, ASR #<shift>`	Allowed shifts 1-32
Rotate right	`Rm, ROR #<shift>`	Allowed shifts 1-31
Rotate right with extend	`Rm, RRX`	

- Examples (<Op2> highlighted in red):
  - `sub r0, r1, #1 @ r0 = r1 - 1`
  - `add r1, r2, r3 @ r1 = r2 + r3`
  - `add r0, r0, r1, lsl #3 @ r0 = r0 + r1 << 3 = r0 + r1*8`
  - `mov r2, r1, asr #1 @ r2 = r1 >> 1 = r1/2`

Building an assembly program

- The C Compiler (gcc or clang) compiles C programs
  - Generates assembly from C
- The Assembler (as or gas) builds assembly programs into machine code
  - Files ending in `.s` are treated as assembly language (notice: capital 'S')
- Task: build and run the asm-test program
  - Then let's look at the source in `render.cpp` and `functions.s`
  - Remember: assembly is CPU-specific. This code must be run on Bela or another ARM based system!

Calling assembly functions from C

- We can write functions in assembly language and call them from C (or C++)
  - Also vice-versa, but we won't cover that here
- To call an assembly language function, we need:
  - Several lines in the `.S` file specifying its name and type

```
.type functionName, %function   @ specifies this is a function
.global functionName            @ means that C files can find it
functionName:                   @ This indicates where the function starts
```

  - A way of returning from the function at the end
    - This is not given to us automatically like using `{ }` in C!
    - Our functions will always end:
      `bx lr`
    - Literally, "branch with exchange, to the link register"
    - `bx` is an instruction; `lr` is a register telling us where we came from
  - Some lines top of the `.S` file specifying the format and syntax of the assembly code (see code examples)

Calling assembly functions from C

- Other things needed to implement a function:
  - A function prototype on in the `.c` or `.cpp` file (or an included `.h` file):

    ```
    int addNumbers(int argumentA, int argumentB);
    ```

    - When declaring a prototype in C++, needs to be inside an `extern "C" {}` block
  - We have 2 arguments which are passed in: `argumentA` and `argumentB`
  - Also have a return value (of type `int`)
- How to pass arguments in assembly:
  - The first four arguments go into `r0`, `r1`, `r2` and `r3`
    - Assuming they are type `int` (or other 32 bit type)
  - More than four arguments: extras go on the stack
    - We won't cover this now
- Returning a value in an assembly function:
  - The return value is always stored in `r0` (assuming `int`)
    - Longer types (e.g. 64 bit `long long`) use two registers (`r0` and `r1`)
  - Store value in `r0` before running `bx lr`

Assembly tasks

- Task 1: using asm-test project
  - Make a new function `subNumbers()` which subtracts `argumentB` from `argumentA`
  - You will need to declare its prototype in `render.cpp`
    - i.e. just like `addNumbers()` is declared now
  - Then implement it in `functions.S`
    - Include the `.type` and `.global` statements right beforehand
  - Which instruction do you use for subtraction?

Assembly tasks

- Task 2: using asm-test project
  - Implement a new function:
    ```
    int addWithShift(int argA, int argB, int argC);
    ```
  - Here is the equivalent in C:
    ```
    return argA + (argB << 1) + (argC >> 1);
    ```
  - Question 1: how do `argA`, `argB` and `argC` appear in our assembly language program?
    - `r0` ↔ `argA`, `r1` ↔ `argB`, `r2` ↔ `argC`
  - Question 2: where do we put the return value?
    - `r0`
  - Question 3: how do we implement the shifts?
    - As part of the add instruction!
    - See the <Operand2> table
  - In `setup()`, change to `z = addWithShift(x, y, 8);`
    - What should answer be?

Playing nice with C

- Remember: the CPU has only one set of registers
  - The C program may have been using them!
  - Don’t assume your assembly function can clobber any of the registers without crashing the program
  - By convention: `r0-r3` and `r12` can be changed by any function without a problem
  - Any other registers you change must be saved at the beginning and restored to their original values at the end!
  - To save registers: push and pop from the stack (a special location in RAM)

```
push {r4, r5, r6, r7}       @ Push registers onto the
push {r8, r10, r11, r14}    @ stack
                            @ Do things in here...
pop {r8, r10, r11, r14}     @ Pop registers back in
pop {r4, r5, r6, r7}        @ reverse order of push
```

Branching

- In assembly, we don't have `if`, `for` or `while`
- Instead, we have branch instructions
  - Think of a branch as a "jump" or "goto" statement
  - Example: `b myLabel`
    - Jumps to the code starting at `myLabel`
  - To declare a label
    - Put on a line by itself (can have a comment afterward)
    - Always followed by a colon
    - By convention, no indentation before a label
- Longer example:

```
  ldr r0, [r7, #0]      @ Load variable from RAM into r0
  b storeResult         @ Branch to label storeResult
  add r0, r0, #1        @ This code never happens
storeResult:
  str r0, [r7, #0]      @ Store back into RAM
```

Branching with conditions

- We can branch only when a given condition is met
  - This is much more powerful, because it lets us add flow control (`if`, etc.) to our program
  - ARM architecture supports 14 different conditions:

Condition Field		
Mnemonic	**Description**	**Description (VFP)**
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS / HS	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
CC / LO	Carry Clear / Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

- Example: `beq myLabel` branches only if last comparison returned an equality

Testing conditions

- We could branch on 14 conditions
  - Do this by appending the condition to the `b` instruction: `beq`, `bne`, `bmi`, `bpl`, ...
  - But what are we actually testing?
    - e.g. `beq` means “branch if equal.” If *what* is equal?
    - There are no registers tested in the branch instruction itself!
- Conditions are based on CPU status flags
  - Special bits that tell us about the previous calculations
  - In other words, testing happens in a previous instruction
  - After the test, in the next instruction, we can branch if the condition is met
- Many compare and test instructions we can use. Some examples:

Compare	Compare		`CMP Rn, <Operand2>`	N Z C V	Update CPSR flags on Rn – Operand2
	negative		`CMN Rn, <Operand2>`	N Z C V	Update CPSR flags on Rn + Operand2
Logical	Test		`TST Rn, <Operand2>`	N Z C	Update CPSR flags on Rn AND Operand2
	Test equivalence		`TEQ Rn, <Operand2>`	N Z C	Update CPSR flags on Rn EOR Operand2

Testing conditions

Compare	Compare		`CMP Rn, <Operand2>`	N	Z	C	V	Update CPSR flags on Rn – Operand2
	negative		`CMN Rn, <Operand2>`	N	Z	C	V	Update CPSR flags on Rn + Operand2
Logical	Test		`TST Rn, <Operand2>`	N	Z	C		Update CPSR flags on Rn AND Operand2
	Test equivalence		`TEQ Rn, <Operand2>`	N	Z	C		Update CPSR flags on Rn EOR Operand2

- Example:

```
    cmp r0, #5              @ Evaluate r0 - 5
    blt isLess              @ Branch if "less than" (r0 < 5)
    add r0, r0, #1          @ This happens only if r0 >= 5
isLess:
```

- We have built an `if()` statement! `if(r0 >= 5) { r0++; }`
  - Important: the form is inverted compared to C
  - If the condition is false, then skip over the part inside the `if()`
- How would we implement an `if` / `else` pair?
  - Hint: you will need another label and a `b` (branch always) statement

Assembly tasks

- Task 3: using asm-test project
  - Make a new function `conditionalNumbers()` which:
    - Subtracts `argumentB` from `argumentA` if `argumentA >= argumentB`; or
    - Adds `argumentA` to `argumentB` if `argumentA < argumentB`
  - You will need to declare its prototype in `render.cpp`
    - i.e. just like `addNumbers()` is declared now
  - Then implement it in `functions.S`
    - Include the `.type` and `.global` statements right beforehand
    - How many labels will you need?

Loops

- How do we implement a loop using branches?
  - First, how would we implement an infinite loop?
  - Branch back to an earlier label

```
loop_start:
  add r0, r0, #1          @ r0++
  b loop_start            @ Go back to beginning, infinitely
```

- How do we implement a conditional loop?
  - e.g. a `for()` or `while()` statement
  - Branch backwards, but using a conditional branch

```
mov r0, #10               @ r0 = 10
loop_start:
  subs r0, r0, #1         @ r0--, updating status flags
  cmp r0, #0              @ No longer needed; subs does it
  bgt loop_start          @ Go back to beginning if r0 > 0
```

  - ARM assembly trick: add an `s` to the end of an arithmetic instruction to update the status flags (saves a compare)

Multiplication

- ARM supports several multiplication instructions
  - Signed, unsigned versions
  - Standard multiply, multiply and accumulate
  - 32 bit versions and smaller versions
  - If you need 64 bits...?
    - Implement it yourself using several 32-bit multiplies
    - This is what C does when you multiply two 64-bit numbers on a 32-bit platform

Operation		§	Assembler	S updates	Action	Notes
Multiply	Multiply		`MUL{S} Rd, Rm, Rs`	N Z C*	Rd := (Rm * Rs)[31:0] (If Rs is Rd, S can be used in Thumb-2)	N, S
	and accumulate		`MLA{S} Rd, Rm, Rs, Rn`	N Z C*	Rd := (Rn + (Rm * Rs))[31:0]	S
	and subtract	T2	`MLS Rd, Rm, Rs, Rn`		Rd := (Rn – (Rm * Rs))[31:0]	
	unsigned long		`UMULL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := unsigned(Rm * Rs)	S
	unsigned accumulate long		`UMLAL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs)	S
	unsigned double accumulate long	6	`UMAAL RdLo, RdHi, Rm, Rs`		RdHi,RdLo := unsigned(RdHi + RdLo + Rm * Rs)	
	Signed multiply long		`SMULL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := signed(Rm * Rs)	S
	and accumulate long		`SMLAL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := signed(RdHi,RdLo + Rm * Rs)	S
	16 * 16 bit	5E	`SMULxy Rd, Rm, Rs`		Rd := Rm[x] * Rs[y]	
	32 * 16 bit	5E	`SMULWy Rd, Rm, Rs`		Rd := (Rm * Rs[y])[47:16]	
	16 * 16 bit and accumulate	5E	`SMLAxy Rd, Rm, Rs, Rn`		Rd := Rn + Rm[x] * Rs[y]	Q
	32 * 16 bit and accumulate	5E	`SMLAWy Rd, Rm, Rs, Rn`		Rd := Rn + (Rm * Rs[y])[47:16]	Q
	16 * 16 bit and accumulate long	5E	`SMLALxy RdLo, RdHi, Rm, Rs`		RdHi,RdLo := RdHi,RdLo + Rm[x] * Rs[y]	
	Dual signed multiply, add	6	`SMUAD{X} Rd, Rm, Rs`		Rd := Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16]	Q
	and accumulate	6	`SMLAD{X} Rd, Rm, Rs, Rn`		Rd := Rn + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16]	Q
	and accumulate long	6	`SMLALD{X} RdLo, RdHi, Rm, Rs`		RdHi,RdLo := RdHi,RdLo + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16]	
	Dual signed multiply, subtract	6	`SMUSD{X} Rd, Rm, Rs`		Rd := Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16]	Q
	and accumulate	6	`SMLSD{X} Rd, Rm, Rs, Rn`		Rd := Rn + Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16]	Q
	and accumulate long	6	`SMLSLD{X} RdLo, RdHi, Rm, Rs`		RdHi,RdLo := RdHi,RdLo + Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16]	
	Signed top word multiply	6	`SMMUL{R} Rd, Rm, Rs`		Rd := (Rm * Rs)[63:32]	
	and accumulate	6	`SMMLA{R} Rd, Rm, Rs, Rn`		Rd := Rn + (Rm * Rs)[63:32]	
	and subtract	6	`SMMLS{R} Rd, Rm, Rs, Rn`		Rd := Rn – (Rm * Rs)[63:32]	
	with internal 40-bit accumulate	XS	`MIA Ac, Rm, Rs`		Ac := Ac + Rm * Rs	
	packed halfword	XS	`MIAPH Ac, Rm, Rs`		Ac := Ac + Rm[15:0] * Rs[15:0] + Rm[31:16] * Rs[31:16]	
	halfword	XS	`MIAxy Ac, Rm, Rs`		Ac := Ac + Rm[x] * Rs[y]	

Multiplication

Operation		§	Assembler	S updates	Action	Notes
Multiply	Multiply		`MUL{S} Rd, Rm, Rs`	N Z C*	Rd := (Rm * Rs)[31:0] (If Rs is Rd, S can be used in Thumb-2)	N, S
	and accumulate		`MLA{S} Rd, Rm, Rs, Rn`	N Z C*	Rd := (Rn + (Rm * Rs))[31:0]	S
	and subtract	T2	`MLS Rd, Rm, Rs, Rn`		Rd := (Rn – (Rm * Rs))[31:0]	
	unsigned long		`UMULL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := unsigned(Rm * Rs)	S
	unsigned accumulate long		`UMLAL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs)	S
	unsigned double accumulate long	6	`UMAAL RdLo, RdHi, Rm, Rs`		RdHi,RdLo := unsigned(RdHi + RdLo + Rm * Rs)	
	Signed multiply long		`SMULL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := signed(Rm * Rs)	S
	and accumulate long		`SMLAL{S} RdLo, RdHi, Rm, Rs`	N Z C* V*	RdHi,RdLo := signed(RdHi,RdLo + Rm * Rs)	S
	16 * 16 bit	5E	`SMULxy Rd, Rm, Rs`		Rd := Rm[x] * Rs[y]	
	32 * 16 bit	5E	`SMULWy Rd, Rm, Rs`		Rd := (Rm * Rs[y])[47:16]	
	16 * 16 bit and accumulate	5E	`SMLAxy Rd, Rm, Rs, Rn`		Rd := Rn + Rm[x] * Rs[y]	Q
	32 * 16 bit and accumulate	5E	`SMLAWy Rd, Rm, Rs, Rn`		Rd := Rn + (Rm * Rs[y])[47:16]	Q
	16 * 16 bit and accumulate long	5E	`SMLALxy RdLo, RdHi, Rm, Rs`		RdHi,RdLo := RdHi,RdLo + Rm[x] * Rs[y]	

- Note a limitation: `Rd != Rm`
  - Valid: `mul r0, r1, r0  @ r0 = r1*r0`
  - Invalid: `mul r0, r0, r1 @ r0 = r0*r1`
- Where is the (integer) divide instruction?
  - There isn't any! At least not ARMv7-A (Cortex-A8)

Assembly coding task

- Task: using asm-factorial project
  - Write the factorial function in assembly
    - Equivalent C code given in `render.cpp`
    - Write your code in `factorial.S`
  - Remember: x! = x * (x-1) * (x-2) * ... * 1
- Hints:
  - Input argument is given to you in `r0`
  - Return value is also in `r0`, so first move r0 elsewhere
    - `mov` statement is useful here
    - `mov` can also be used to put numbers into registers
    - Remember that you can modify `r0` to `r3` without consequence
  - You will need to write a conditional loop
    - Move `r0` to a different register, use `subs` on this register to count down toward 0

ARM floating point

- All instructions so far have been for integer data
  - `add`, `sub`, `mul`, `mla`, etc. are all for 32-bit int types on Cortex-A8
  - Can make longer (64-bit) int calculations out of several instructions, but doesn't natively work on `float/double`
- ARM Cortex-A8 has two different FPUs
- Vector Floating Point (VFP)
  - Classic hardware FPU for single and double precision
    - Despite the name, not actually used for vector calculations
  - On Cortex-A8, it is the VFPv3Lite version
    - This means it uses the v3 instruction set, but implemented in a way that saves chip area but is slower than full VFPv3
- NEON Advanced SIMD extension
  - Single Instruction, Multiple Data
  - True vector-based calculations for `int` and single-precision `float` data
    - up to 128 bits per instruction = 4 floats

Using VFP

- `gcc` (and typically `clang`) uses VFP by default for all `floats` and `doubles`
  - VFP is the only hardware option for `double`
- VFP has its own instruction set (see companion materials)

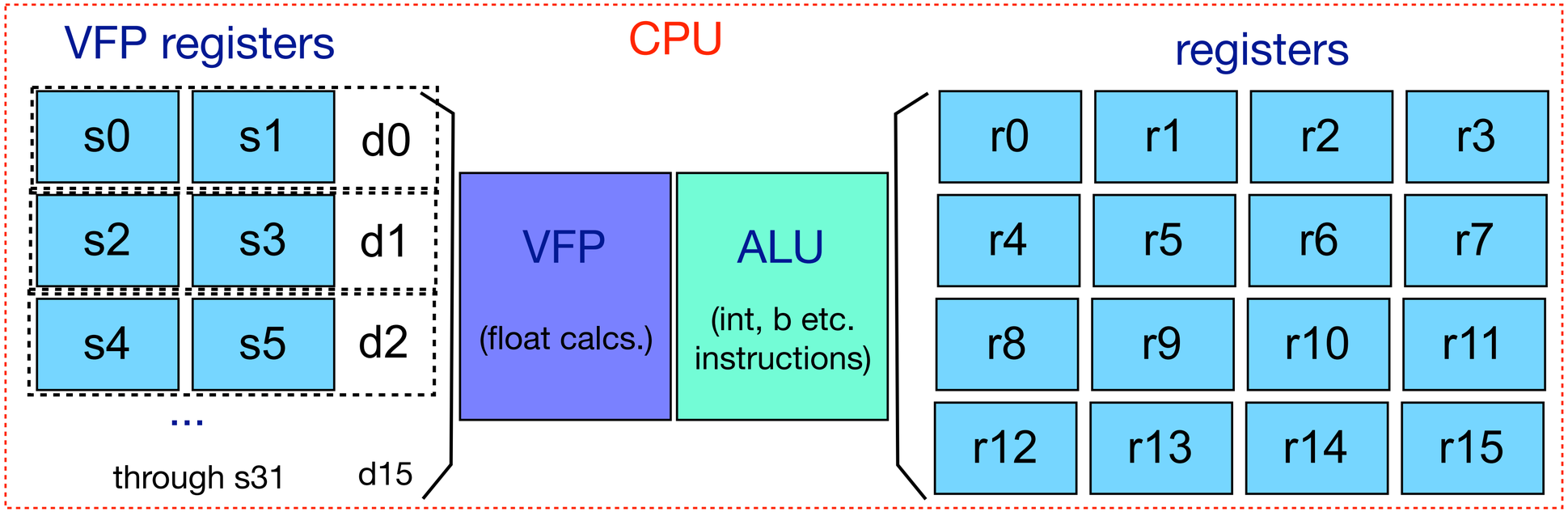
Operation		§	Assembler	Exceptions	Action
Vector arithmetic	Multiply		VMUL{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fn * Fm
	and negate		VNMUL{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := – (Fn * Fm)
	and accumulate		VMLA{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fd + (Fn * Fm)
	negate and accumulate		VMLS{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fd – (Fn * Fm)
	and subtract		VNMLS{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := – Fd + (Fn * Fm)
	negate and subtract		VNMLA{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := – Fd – (Fn * Fm)
	Add		VADD{C}.<P> Fd, Fn, Fm	IO, OF, IX	Fd := Fn + Fm
	Subtract		VSUB{C}.<P> Fd, Fn, Fm	IO, OF, IX	Fd := Fn – Fm
	Divide		VDIV{C}.<P> Fd, Fn, Fm	IO, DZ, OF, UF, IX	Fd := Fn / Fm
	Absolute		VABS{C}.<P> Fd, Fm		Fd := abs(Fm)
	Negative		VNEG{C}.<P> Fd, Fm		Fd := – Fm
	Square root		VSQRT{C}.<P> Fd, Fm	IO, IX	Fd := sqrt(Fm)

.f32 or .f64

  - Yes, we now have a division instruction (unlike for `int`)
  - Multiply and accumulate with several different signs
  - Also a square root instruction
  - Like other ARM instructions, every VFP instruction can be conditional (e.g. `vmuleq` = multiply only if last test was equal)

VFP registers

- The VFP unit has its own registers
  - Entirely separate from ARM registers `r0-r15`
  - VFP has 32 single-precision (32-bit) registers named `s0-s31`
  - These can also be treated as 16 double-precision (64-bit) registers `d0-d15`



  - VFPv3 (e.g. on Bela) adds 16 more double-precision registers `d16-d31` (but not `s32-s63`)

Passing floating point arguments

- When calling assembly functions from C:
  - Normally, arguments in `r0-r3`; return value in `r0`
  - What if we are passing or returning float values?
  - Depends on Linux *ABI* we use
    - ABI = Application Binary Interface
    - Agreed-upon standard shared by the OS and all programs
  - In standard (`softfp`) ABI, floating-point arguments come in on ARM registers `r0-r3`, return in `r0`, just like any other function
- Thus we have to move from ARM to VFP and back
  - For example: `float myFunction(float arg1, float arg2);`

```
myFunction:
  vmov s0, r0          @ s0 = r0 (arg1)
  vmov s1, r1          @ s1 = r1 (arg2)
                       @ do stuff here...
  vmov r0, s0          @ r0 = s0 (return value)
```

Passing floating point arguments

```
myFunction:
  vmov s0, r0          @ s0 = r0 (arg1)
  vmov s1, r1          @ s1 = r1 (arg2)
                       @ do stuff here...
  vmov r0, s0          @ r0 = s0 (return value)
```

- Moving from ARM (e.g. `r0`) to VFP (e.g. `s0`) is fast
- Moving from VFP to ARM is slow
  - Takes 20+ cycles just to move the value between registers!
  - If you return `float` values many times per second, this is a big performance hit
- Alternative (which Bela uses): hard float (armhf) ABI
  - Float arguments come in directly on VFP registers
    - Inputs on `s0-s3`, return on `s0` (assuming all float values)
    - Mixed `int` and `float` arguments use both ARM and VFP registers
  - Saves time: no extra moves needed
  - But armhf code can't run at all on systems without an FPU

Floating point task

- Task: using asm-float project
  - Implement the Euclidean distance function in assembly language:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

  - Use the subtraction, multiplication and square root instructions

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources