Assembling Mathematics

Steven R. Bagley

Stack Frames

- Data stored on the stack as part of a function call forms part of the *stack frame* for that invocation
- Stack frames are used to store register values, but also to create space for local variables used within the function
- Also used to preserve the link register (`R14`)

Local variables go on the stack because you can then be sure of a unique instance of them for that invocation. Functions may be called while the function is already running (e.g. recursion, multi-threaded code, etc.) Can’t always just be stored in registers (Even if we have enough of them, since we might need to pass the address to something)

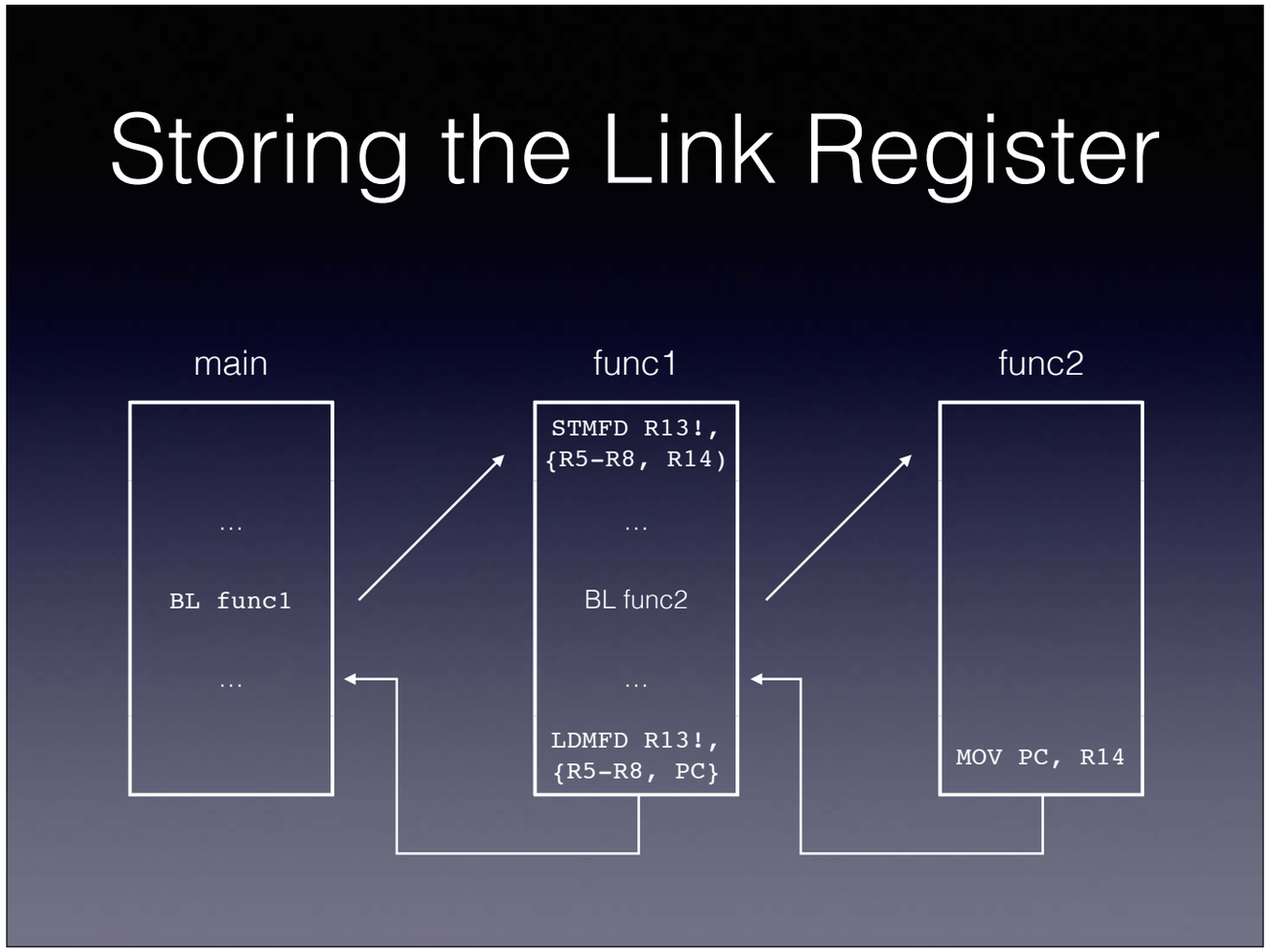
Stack Frames

- When a procedure exits and return to the caller, everything it put on the stack must be popped
- This is why local variables vanish once a function exits
- The caller expects the stack to be exactly as it left it

Storing the Link Register

- If we are a leaf function (i.e. we don't call anything else) then there is no need to store the link register
- If we do call a function, then it is necessary to preserve the link register
- Why?

Storing the Link Register



Note how we restore to PC, not R14 — saves us an instruction

Storing the Link Register

- The `BL func1` in main stores the return address in `R14`
- But then the `BL func2` inside `func1` overwrites it
- So `func2` returns correctly to `func1`
- But if `func1` were to return using `MOV PC, R14` then `R14` would have the wrong value

Storing the Link Register

- Non-leaf functions definitely need to stack the link register value
- But note the stack frame push/pop in `func1`
- The `LDMFD` restores the stored return address directly into the `PC`
- This causes an instantaneous return to `main`

Local Variables

- Local variables are stored on the stack
- Guarantees a unique instance of the variables for each function invocation
- Can easily create space for them on the stack by using a `SUB` instruction after we preserve the registers
- Use an `ADD` later to remove them all quickly…
- Can then use an offset from `R13` to access them
- But remember the offset will change as you push and pop more values…

Recursion

- Spent some time considering how to implement functions in ARM assembler
- Use `BL` to jump to the function
- Stores the return address in `R14`
- Use `MOV PC, R14` to return
- Use the stack to preserve registers (including `R14` if we need to call another function)

Recursion

- This mechanism also works to implement recursive functions
- Functions that call themselves
- Has to be away to escape the recursion, otherwise it will continue forever
- Or at least until the stack runs out…

Factorial

- One classic example of recursion is the factorial function
- This is defined in terms of itself
  factorial(*n*) = *n* x factorial(*n*-1)
  factorial(0) = 1
- Thus, factorial(4) = 4 x 3 x 2 x 1 x 1 = 24

```
int factorial(int n)
{
    if(n == 0)
    {
        return 1;
    }
    else
    {
        return n * factorial(n-1);
    }
}
```

Easy to implement in C. How about in ARM assembler?

Recursion in ARM

- Let's look at how to do recursion in ARM assembler
- Implement `factorial()` because it is simple
- Probably be thankful that the C compiler takes care of it for us
- Note that `factorial()` is *not* tail-recursive, we do the multiply *after* we call `factorial()`

See also the 'Notes on Recursion' handout on the website

Recursion in ARM

- Will need to store the value of `n` passed into the function
- Store it on the stack along with the link register `R14`
- Need to make sure that we can cope with being called both recursively and also the final case
- For `factorial()`, when `n` reaches the value of `0`
- As always, we'll pass input in `R0` and return in `R0`

Go and implement this…

Factorial Stack Frame

Stack frame for factorial

Final words on Stacks

- Note that the `factorial()` stack contains many different instances of `n`
- Should be obvious why we have to store `n` on the stack
- It's the only place that's guaranteed to be unique
- The compiler's job to generate the correct code for the stack frame

Or ours if we are coding in machine code…

Multiplication

- Used the `MUL` instruction to multiply two numbers together
- The multiply instructions were not part of the original ARM CPU
- Added to the instruction set for ARM2
- Had to be squeeze into the bit patterns so don't appear as normal data processing instructions…

ARM2 was the first version that was used in anger, in the Acorn Archimedes line…

MUL instruction

- Mnemonic: `MUL`
- Three operands: Destination register, and *two* source registers
- Cannot multiply by an immediate value
- Destination is filled with the result of multiplying the two source registers

Again, an S suffix means they instruction will set the condition codes

MLA instruction

- Mnemonic: `MLA`
- Combined Multiply and Add
- Takes *four* operands: Destination register, and *two* source registers to be multiplied and a final register to be added on
- Destination is filled with the result of multiplying the first two source registers and adding the third

Again, an S suffix means they instruction will set the condition codes

```
MUL R0, R0, R0          ; R0 = R0 * R0

MOV R2, #5
MUL R1, R0, R2          ; R1 = R0 * 5


MLA R0, R1, R2, R3      ; R0 = R1 * R2 + R3
```

Note must be registers

Do some demos

Multiplication

- Noted earlier that the original ARM1 CPU did not have any support for multiplication
- Still no support for division
- Relatively easy to write software that can multiply and divide
- By using a combination of adds and shifts
- To understand this need to think about how we do multiplication *by hand*

And I don’t mean on a calculator…

Multiplication

- The product of *m* and *n* digit integers requires (at most) *m* + *n* digits
- So, multiplying two 4-digit numbers requires 8 digits
- Let's look at some examples…

Talk through the process on paper…

Start with multiplying by a single digit

Then do two four digit numbers

Multiplication

					6	2	9	5	
x					2	8	1	7	
				4	4	0	6	5	
				6	2	9	5		
			5	0	3	6	0		
+	1	2	5	9	0				Partial Sums
=	1	7	7	3	3	0	1	5	

Not how each line of the multiplications is shifted to the left by one column

Really obvious on line two of the partial sums where we multiply by one

Binary Multiplication

- The same process holds true for multiplying two binary numbers
- Product of *m* and *n*-bit integers requires *m*+*n* bits
- Again, we use a pattern of shifts and adds
- This time each shift is a multiplication by two, not ten

Again do some examples…

Binary Multiplication

					1	1	0	1	$a = 13_{10}$
x					1	0	1	1	$a = 11_{10}$
					1	1	0	1	
				1	1	0	1		
			0	0	0	0			
+		1	1	0	1				Partial Sums
=	1	0	0	0	1	1	1	1	$c = 143_{10}$

Not how each line of the multiplications is shifted to the left by one column

Because each bit is either one or zero, each partial sum is either same digits shifted to the left, or zero

n-bit Binary Multiplication

- Can easily implement this algorithm in computer code
- Using a combination of shifts and adds
- Use >> and << bitwise shift operators in C
- Or `LSL` and `LSR` in ARM

Show C then implement in ARM

```
c = 0;
for(i = 1; i < n; i++)
{
    if((b & 1) != 0)
    {
        c = c + a;
    }
    b = b >> 1;
    a = a << 1;

}
```

Calculates c = a * b, where n denotes how many bits are used

Note that we use AND to mask off the LSB of b, if its one we add a

because we shift b to the right we end up looking at every bit of b

Shift a to the left to visit each positiion

Binary Multiplication

- Can optimise it slightly by stopping when `b == 0` rather than doing the loop n times
- Will technically produce a 2n-bit result (so multiplying two 8-bit numbers will produce a 16-bit result)
- Although the `MUL` instruction only produces 32-bit results

Signed Multiplication

- Routine as it stands doesn't work for signed numbers
- But can be adapted to work
  - Could check the signs
  - Convert to unsigned
  - Multiply
  - Then fix sign

Signed Multiplication

- The alternative is that we first convert the *n*-bit signed integers to *2n*-bit signed integers
- To do this, we need to sign extend them…
- Signed integers can be infinitely sign-extended
- Positive integers can be prefixed by an infinite number of 0s
- Negative integers can be prefixed by an infinite number of 1s

I.e. just copy the sign bit to the left

Signed Multiplication

- Sign-extend *a* and *b* to *2n* digits for multiplication
- Use the same multiplication algorithm, but it must loop for 2n times
- But can still exit early when `b == 0`

b will only equal zero for positive numbers

Signed Multiplication

	1	1	1	1	1	1	0	1	a = -3_{10}
x	0	0	0	0	0	1	0	1	b = 5_{10}
	1	1	1	1	1	1	0	1	
	0	0	0	0	0	0	0		
+	1	1	1	1	0	1			Partial Sums
=	1	1	1	1	0	0	0	1	c = -15_{10}

Not how each line of the multiplications is shifted to the left by one column

Because each bit is either one or zero, each partial sum is either same digits shifted to the left, or zero

Signed Multiplication

	1	1	1	1	1	1	0	1	a = -3_{10}
x	1	1	1	1	1	0	1	1	b = -5_{10}
	1	1	1	1	1	1	0	1	
	1	1	1	1	1	0	1		
	0	0	0	0	0	0			
+	1	1	1	0	1				Partial Sums
=	1	1	0	1	1	1	1	1	c = -33_{10}

Not how each line of the multiplications is shifted to the left by one column

Because each bit is either one or zero, each partial sum is either same digits shifted to the left, or zero

Signed Multiplication

	1	1	1	1	1	1	0	1	a = -3_{10}
x	1	1	1	1	1	0	1	1	b = -5_{10}
	1	1	1	1	1	1	0	1	
	1	1	1	1	1	0	1		
	0	0	0	0	0	0			
+	1	1	1	0	1				Partial Sums
=	1	1	0	1	1	1	1	1	c = -3_{10}

Not how each line of the multiplications is shifted to the left by one column

Because each bit is either one or zero, each partial sum is either same digits shifted to the left, or zero

Signed Multiplication

	1	1	1	1	1	1	0	1	a = -3_{10}
x	1	1	1	1	1	0	1	1	b = -5_{10}
	1	1	0	1	1	1	1	1	c = -33_{10} (partial)
	1	1	0	1					
	1	0	1						
	0	1							
+	1								Partial Sums
=	0	0	0	0	1	1	1	1	c = 15_{10}

Not how each line of the multiplications is shifted to the left by one column

Because each bit is either one or zero, each partial sum is either same digits shifted to the left, or zero

Long Division

```
Divisor  0  0  2  1  3   Quotient
+ 8 1 | 1  7  3  3  0   Dividend
      8  1  0  0  0  0   x0
         8  1  0  0  0   x0
            8  1  0  0   x2
      -  1  6  2  0  0
         ------------
            1  1  3  0
               8  1  0   x1
         -     8  1  0
         ------------
               3  2  0
                  8  1   x3
         -     2  4  3
         ------------
                  7  7   Remainder
```

works in pretty much in the reverse of multiplication

Long Division in Binary

```
Divisor  0 0 1 0 1 1 1  Quotient
+ 1 0 1 | 1 1 1 0 1 0 1  Dividend
        1 0 1 0 0 0 0 0  x0
          1 0 1 0 0 0 0  x0
        - 1 0 1 0 0 0 0  x1
          ---------
            1 0 0 1 0 1
            1 0 1 0 0 0  x0
          - 1 0 1 0 0    x1
          -------
              1 0 0 0 1
            - 1 0 1 0    x1
            -----------
                1 1 1
              1 0 1      x1
              -------
                  1 0    Remainder
```

Again the same in binary

n-bit Binary Division

- This can be implanted as an algorithm given a dividend `a` and a divisor `b`
- Calculates their quotient `d = a / b`
- Leaves the remains in `a = a % b`
- `n` is the desired number of bits

```
d = 0;
b = b << n;
for(i = 1; i < n; i++)
{
    b = b >> 1;
    d = d << 1;
    if(a > b)
    {
        a = a - b;
        d = d + 1;
    }
}
```

Calculates c = a * b, where n denotes how many bits are used

Note that we use AND to mask off the LSB of b, if its one we add a

because we shift b to the right we end up looking at every bit of b

Shift a to the left to visit each positiion

Integer Division

- Integer division is rather tricky. This routine only handles positive integers
- Each stage does a binary subtraction to yield a new dividend
- Initially position divisor so it is just bigger than dividend
- Then start the subtract-and-shift-right process
- Anything non-zero that is left is the integer remainder

Integer Division

- Need to be careful with the size of dividend and divisor
- Need to be able to shift the divisor completely to the left of the dividend
- So realistically using 32-bit registers, we can only divide a 16-bit number by another 16-bit number
- Can do 32-bit division but have to use multiple registers

Integer Division

- If you need to divide by a power of two
- It's much easier (and faster) to use an arithmetic shift right
- Can sometimes even be combined into an **`ADD`**/**`SUB`** instruction

```
MOV R0, R0 ASR #1      ; Divide by 2^1 (or 2)
MOV R0, R0 ASR #2      ; Divides by 2^2 (or 4)
MOV R3, R0 ASR #8      ; Divides by 2^8 (or 256)

ADD R3, R2, R1 ASR #8 ; R3 = R2 + R1/8
RSB R5, R3, R1 ASR #2 ; R5 = R1/4 - R3
```

Remember that ASR keeps the sign bit set…

Notes on Maths in ARM

- ARM instructions only do very basic operations
- Need to chain them together to build up more complex things
- This may involve using and re-using registers to store temporary values
- Also, worth remembering the BODMAS precedence rules
- Do the higher precedence things before the lower precedence

```
x = a * b - c * d;

LDR R0, _a
LDR R1, _b
MUL R0, R0, R1  ; Performs a * b

LDR R1, _c      ; Can reuse R1 now
LDR R2, _d
MUL R1, R1, R2  ; Performs c * d

SUB R0, R0, R1  ; Performs the subtraction
```

On the other hand, if we needed the values of a or b elsewhere in the program we may well chose to put the result in a different register so we don’t have to fetch it from memory again…

```
x = a + b - c - d;

LDR R0, _a
LDR R1, _b
ADD R0, R0, R1  ; Performs a + b

LDR R1, _c      ; Can reuse R1 now
SUB R0, R0, R1
LDR R1, _d
SUB R0, R0, R1
```

If you have things at the same level of precedence then its much easier to reuse registers

Real Numbers

- Spent our time looking at how the computer handles integers
- But what about numbers such as
  - 3.1415926
  - 1/3 = 0.33333333
- How do we represent quantities (numerically) smaller than 1?

Shifting the Point

- We use the *decimal point* to separate integer and fractional parts
- Digits after the point denote 1/10th, 1/100th, …
- Can use a *binary point* in a similar way

Bit	4^{th}	3^{rd}	2^{nd}	1^{st}	0th	•	-1^{st}	-2^{nd}	-3^{rd}	-4^{th}
Weight	2^4 16	2^4 8	2^4 4	2^4 2	2^4 1		2^4 1/2	2^4 1/4	2^4 1/8	2^4 1/16

Shifting the Point

Bit	4th	3rd	2nd	1st	0th	•	-1st	-2nd	-3rd	-4th
Weight	2^4 16	2^4 8	2^4 4	2^4 2	2^4 1		2^4 1/2	2^4 1/4	2^4 1/8	2^4 1/16

- So 0.101_2 in decimal would be $2^{-1} + 2^{-3} = 0.5 + 0.125 = 0.625$
- What would 0.1_{10} be in binary?

Go calculate this on the visualiser

Shifting the Point

Bit	4th	3rd	2nd	1st	0th	•	-1st	-2nd	-3rd	-4th
Weight	2^4 16	2^4 8	2^4 4	2^4 2	2^4 1		2^4 1/2	2^4 1/4	2^4 1/8	2^4 1/16

- What would 0.1_{10} be in binary?
  0.0001100110011…2
- Digits repeat for ever — there's no exact binary representation

Go calculate this on the visualiser

Order	Fraction
0	1
-1	0.5
-2	0.25
-3	0.125
-4	0.0625
-5	0.03125
-6	0.015625
-7	0.0078125
-8	0.00390625
-9	0.001953125
-10	0.0009765625
-11	0.00048828125
-12	0.000244140625
-13	0.0001220703125
-14	0.00006103515625
-15	0.000030517578125
-16	0.0000152587890625

Negative powers of 2 as decimals

Fixed Point Arithmetic

- How do we represent real numbers on a computer?
- One mechanism that is used is fixed-point arithmetic
- Moves the ‘binary point’ as it were, so some bits represent positive powers and others negative powers

Fixed-Point Arithmetic

	MSB							LSB
Bit Position	7	6	5	4	3	2	1	0
Order	3^{rd}	2^{nd}	1^{st}	0th	-1^{st}	-2^{nd}	-3^{rd}	-4^{th}
Weight	2^4 8	2^4 4	2^4 2	2^4 1	2^4 1/2	2^4 1/4	2^4 1/8	2^4 1/16

Fixed-Point Arithmetic

- Another way to consider it is that the binary number now always represents the numerator in a fraction
- Where the denominator is always a power of 2
- So in the previous example, the denominator would be 16
- We then count in steps of $1/16^{th}$ instead of 1

Fixed-Point Arithmetic

- Addition and Subtraction of fixed-point numbers works as normal
- We are just adding fractions (although this means that they must be using the same denominator)
- Multiplication works too, but we must scale afterwards…

Remember multiplying two fractions multiples the two numerators and the two denominators, so we need to scale the denominator down…

Fixed-Point Arithmetic

- Easy to convert between fixed-point and integer numbers using a bit shift
- Shift to the left by the number of fractional bits to convert an integer to fixed-point
- Or Shift to the right by the number of bits used for the fractional part to convert to an integer
- This will always round down, but easy enough to make it round up too

Fixed-Point Overview

- Fixed-point arithmetic is effectively still integer arithmetic as far as the CPU is concerned
- So still very fast
- Often used in games, DSP etc to handle real numbers
- No need for hardware support

Fixed-Point Overview

- Has a limited range
- Depending on how many bits are assigned to the integer and fractional part
- E.g. the common 16:16 fixed-point format can only represent numbers between ±32768
- Insufficient for constants used in scientific calculations
- Also, can only store exact representations of multiples of powers of two
- No way to represent say 0.01 for a financial calculations

16-bits for both the integer and real parts

Scientific Notation

- Use scientific notation for very large/small numbers, e.g 2.998×10^{9}, 6.626×10^{-34}
- Normalizes all numbers into a standard format