Introduction to the Cortex-M4

Matthias J. Kannwischer and Bo-Yin Yang
Academia Sinica, Taipei, Taiwan
matthias@kannwischer.eu

08 June 2023, Summer School on real-world crypto and privacy, Vodice, Croatia

Recap: why program in assembly

- Compilers are useful, but not that ‘good’
- Assembly gives precise control
- Can be critical for a secure implementation!
  - Constant-time
  - Correct order of instructions with masking
- Can be critical for a fast implementation!

Recap: why program in assembly

- Compilers are useful, but not that ‘good’
- Assembly gives precise control
- Can be critical for a secure implementation!
  - Constant-time
  - Correct order of instructions with masking
- Can be critical for a fast implementation!

Recap: why program in assembly

- Compilers are useful, but not that ‘good’
- Assembly gives precise control
- Can be critical for a secure implementation!
  - Constant-time
  - Correct order of instructions with masking
- Can be critical for a fast implementation!

Recap: why program in assembly

- Compilers are useful, but not that ‘good’
- Assembly gives precise control
- Can be critical for a secure implementation!
  - Constant-time
  - Correct order of instructions with masking
- Can be critical for a fast implementation!

Our platform: Arm

- Arm company designs CPUs, does not build them
- Market leader for mobile devices, embedded systems

- Armv7E-M architecture
- Cortex-M4 implements this architecture
- Released in 2010, widely deployed

- STM32F407VGT6
  - Cortex-M4 + peripherals
- 1024 KB flash
- 192 KB SRAM
- 168 MHz CPU

Our platform: Arm

- Arm company designs CPUs, does not build them
- Market leader for mobile devices, embedded systems

- Armv7E-M architecture
- Cortex-M4 implements this architecture
- Released in 2010, widely deployed

- STM32F407VGT6
  - Cortex-M4 + peripherals
- 1024 KB flash
- 192 KB SRAM
- 168 MHz CPU

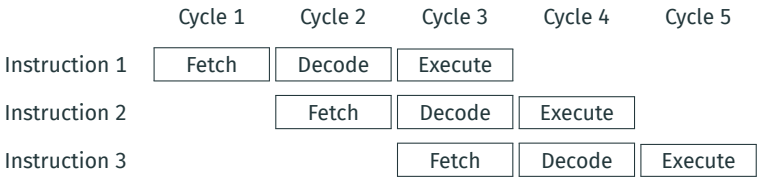
Pipeline

- Cortex-M4 has pipelined execution
- 3 stages: fetch, decode, execute

- Branching breaks this
  - But remedied by branch prediction + speculative execution
- Execute happens in one cycle: dependencies do not cause stalls

Pipeline

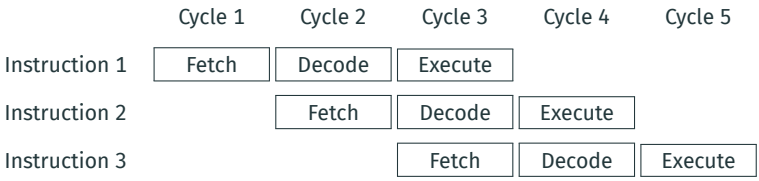
- Cortex-M4 has pipelined execution
- 3 stages: fetch, decode, execute



- Branching breaks this
  - But remedied by branch prediction + speculative execution
- Execute happens in one cycle: dependencies do not cause stalls

Pipeline

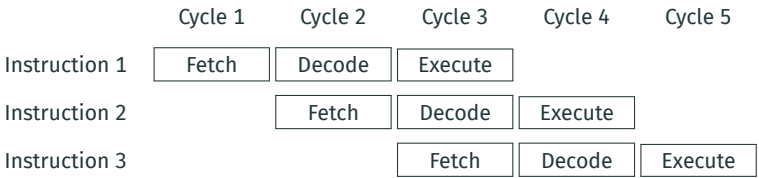
- Cortex-M4 has pipelined execution
- 3 stages: fetch, decode, execute



- Branching breaks this
  - But remedied by branch prediction + speculative execution
- Execute happens in one cycle: dependencies do not cause stalls

Pipeline

- Cortex-M4 has pipelined execution
- 3 stages: fetch, decode, execute



- Branching breaks this
  - But remedied by branch prediction + speculative execution
- Execute happens in one cycle: dependencies do not cause stalls

Registers

- 16 registers: `r0`–`r15`
- Some special registers
  - `r13`: `sp` (stack pointer)
  - `r14`: `lr` (link register)
  - `r15`: `pc` (program counter)
- `r0`–`r12` are general purpose and can be freely used
- `r14` can also be freely used after being saved to the stack

Registers

- 16 registers: `r0`–`r15`
- Some special registers
  - `r13`: `sp` (stack pointer)
  - `r14`: `lr` (link register)
  - `r15`: `pc` (program counter)
- `r0`–`r12` are general purpose and can be freely used
- `r14` can also be freely used after being saved to the stack

Registers

- 16 registers: `r0`–`r15`
- Some special registers
  - `r13`: `sp` (stack pointer)
  - `r14`: `lr` (link register)
  - `r15`: `pc` (program counter)
- `r0`–`r12` are general purpose and can be freely used
- `r14` can also be freely used after being saved to the stack

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - By default, flags never get updated!
  - Many instructions have a variant that sets flags by appending `s`
- Bitwise operations: `eor`, `and`, `orr`, `mvn`, `orn`, `bic`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant ('immediate')

Combined barrel shifter

- Distinctive feature of ARM architecture
- Every `Rm` operand goes through barrel shifter
- Possible to do this: `eor r0, r1, r2, lsl #2`
- Two instructions for the price of one, only costs 1 cycle
- Optimized code uses this all the time
- Possible with most arithmetic instructions

Combined barrel shifter

- Distinctive feature of ARM architecture
- Every `Rm` operand goes through barrel shifter
- Possible to do this: `eor r0, r1, r2, lsl #2`
- Two instructions for the price of one, only costs 1 cycle
- Optimized code uses this all the time
- Possible with most arithmetic instructions

Combined barrel shifter

- Distinctive feature of ARM architecture
- Every `Rm` operand goes through barrel shifter
- Possible to do this: `eor r0, r1, r2, lsl #2`
- Two instructions for the price of one, only costs 1 cycle
- Optimized code uses this all the time
- Possible with most arithmetic instructions

Barrel shifter example

Example:

```
mov r0, #42
mov r1, #37
ror r1, r1, #1
orr r2, r0, r1
lsl r2, r2, #1
eor r0, r2
```

More efficient:

```
mov r0, #42
mov r1, #37
orr r2, r0, r1, ror #1
eor r0, r0, r2, lsl #1
```

- Barrel shifter does not update `Rm`, i.e. `r1` and `r2`!
- A very common use is as a mask with `Rm, asr #31`!

Barrel shifter example

Example:

```
mov r0, #42
mov r1, #37
ror r1, r1, #1
orr r2, r0, r1
lsl r2, r2, #1
eor r0, r2
```

More efficient:

```
mov r0, #42
mov r1, #37
orr r2, r0, r1, ror #1
eor r0, r0, r2, lsl #1
```

- Barrel shifter does not update `Rm`, i.e. `r1` and `r2`!
- A very common use is as a mask with `Rm, asr #31`!

Barrel shifter example

Example:

```
mov r0, #42
mov r1, #37
ror r1, r1, #1
orr r2, r0, r1
lsl r2, r2, #1
eor r0, r2
```

More efficient:

```
mov r0, #42
mov r1, #37
orr r2, r0, r1, ror #1
eor r0, r0, r2, lsl #1
```

- Barrel shifter does not update `Rm`, i.e. `r1` and `r2`!
- A very common use is as a mask with `Rm, asr #31`!

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels
- Assembler and linker later resolve the address ‘

```
mov r0, #42
b somelabel
mov r0, #37
somelabel:
...
```

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels
- Assembler and linker later resolve the address ‘

```
mov r0, #42
b somelabel
mov r0, #37
somelabel:
...
```

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels
- Assembler and linker later resolve the address ‘

```
mov r0, #42
b somelabel
mov r0, #37
somelabel:
...
```

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels
- Assembler and linker later resolve the address

```
mov r0, #42
b somelabel
mov r0, #37
somelabel:
...
```

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels
- Assembler and linker later resolve the address ‘

```
mov r0, #42
b somelabel
mov r0, #37
somelabel:
...
```

Conditional branches

- How to do a `for`/`while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - And many more

Conditional branches

- How to do a `for`/`while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - And many more

Conditional branches

- How to do a `for`/`while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - And many more

Conditional branches

- How to do a `for`/`while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - And many more

Conditional branches

- How to do a `for`/`while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - And many more

Conditional branches

- How to do a `for`/`while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - And many more

Conditional branches

- How to do a `for`/`while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - And many more

Conditional branches (example)

In C:

```
uint32_t a, b = 100;

for (a = 0; a <= 50; a++) {
    b += a;
}
```

In asm:

```
mov r0, #0     // a
mov r1, #100   // b

loop:
add r1, r0     // b += a

add r0, #1     // a++
cmp r0, #50    // compare a and 50
bls loop       // loop if <=
```

Conditional Execution

- Instructions can be executed conditionally when they are part of an `IT` block
- For Example,

```
cmp r0, #42
ITE eq
addeq r1, r1, r2
subne r1, r1, r2
```

- Will add `r2` to `r1` if `r0` is equal to 42; otherwise it will subtract `r2` from `r1`

Conditional Execution

- Instructions can be executed conditionally when they are part of an `IT` block
- For Example,

```
cmp r0, #42
ITE eq
addeq r1, r1, r2
subne r1, r1, r2
```

- Will add `r2` to `r1` if `r0` is equal to 42; otherwise it will subtract `r2` from `r1`

Conditional Execution (2)

- All instructions will be executed; if condition is not satisfied the result will be discarded
  - Instructions for which the condition is not satisfied act as a `nop`
  - This implies that secret conditions result in constant-time as long as there is no branch instruction inside of the `IT` block
- Block can consist of up to four instructions
- First instruction always needs to be in the *then* (`T`) branch; for the rest it arbitrary
  - Examples: `IT`, `ITT`, `ITTTT`, `ITETE`
  - The *then* condition needs to match the condition in the `IT` instruction
  - the *else* (`E`) conditions need to be the opposite

Conditional Execution (2)

- All instructions will be executed; if condition is not satisfied the result will be discarded
  - Instructions for which the condition is not satisfied act as a `nop`
  - This implies that secret conditions result in constant-time as long as there is no branch instruction inside of the `IT` block
- Block can consist of up to four instructions
- First instruction always needs to be in the *then* (`T`) branch; for the rest it arbitrary
  - Examples: `IT`, `ITT`, `ITTTT`, `ITETE`
  - The *then* condition needs to match the condition in the `IT` instruction
  - the *else* (`E`) conditions need to be the opposite

The stack

- Often data does not fit in registers
- Solution: push intermediate values to the stack (changes `sp`)
- `push {r0, r1}`
- Can now re-use `r0` and `r1`
- Later retrieve values in any register you like: `pop {r0, r2}`

- Can load from the stack without moving `sp`
- Not popping all pushed values will crash the program

The stack

- Often data does not fit in registers
- Solution: push intermediate values to the stack (changes `sp`)
- `push {r0, r1}`
- Can now re-use `r0` and `r1`
- Later retrieve values in any register you like: `pop {r0, r2}`
- Can load from the stack without moving `sp`
- Not popping all pushed values will crash the program

The stack

- Often data does not fit in registers
- Solution: push intermediate values to the stack (changes `sp`)
- `push {r0, r1}`
- Can now re-use `r0` and `r1`
- Later retrieve values in any register you like: `pop {r0, r2}`

- Can load from the stack without moving `sp`
- Not popping all pushed values will crash the program

The stack

- Often data does not fit in registers
- Solution: push intermediate values to the stack (changes `sp`)
- `push {r0, r1}`
- Can now re-use `r0` and `r1`
- Later retrieve values in any register you like: `pop {r0, r2}`
- Can load from the stack without moving `sp`
- Not popping all pushed values will crash the program

The stack

- Often data does not fit in registers
- Solution: push intermediate values to the stack (changes `sp`)
- `push {r0, r1}`
- Can now re-use `r0` and `r1`
- Later retrieve values in any register you like: `pop {r0, r2}`

- Can load from the stack without moving `sp`
- Not popping all pushed values will crash the program

The stack

- Often data does not fit in registers
- Solution: push intermediate values to the stack (changes `sp`)
- `push {r0, r1}`
- Can now re-use `r0` and `r1`
- Later retrieve values in any register you like: `pop {r0, r2}`

- Can load from the stack without moving `sp`
- Not popping all pushed values will crash the program

Memory

- Stack is nice for intermediate values, but not for constants or lookup tables
- '*word*' = 32 bit, '*halfword*' = 16 bit, '*doubleword*' = 64 bit, '*byte*' = 8 bit, '*nibble*' = 4 bit
- Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- Ends up *somewhere* in RAM, need a label to access it
- For *n* bytes of uninitialized memory, use a label and `.skip n`
  - For *n* bytes of 0-initialized data, use `.lcomm somelabel, n`
- For global constants in ROM/flash, use `.section .rodata`

Memory

- Stack is nice for intermediate values, but not for constants or lookup tables
- '*word*' = 32 bit, '*halfword*' = 16 bit, '*doubleword*' = 64 bit, '*byte*' = 8 bit, '*nibble*' = 4 bit
- Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- Ends up *somewhere* in RAM, need a label to access it
- For *n* bytes of uninitialized memory, use a label and `.skip n`
  - For *n* bytes of 0-initialized data, use `.lcomm somelabel, n`
- For global constants in ROM/flash, use `.section .rodata`

Memory

- Stack is nice for intermediate values, but not for constants or lookup tables
- '*word*' = 32 bit, '*halfword*' = 16 bit, '*doubleword*' = 64 bit, '*byte*' = 8 bit, '*nibble*' = 4 bit
- Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- Ends up *somewhere* in RAM, need a label to access it
- For *n* bytes of uninitialized memory, use a label and `.skip n`
  - For *n* bytes of 0-initialized data, use `.lcomm somelabel, n`
- For global constants in ROM/flash, use `.section .rodata`

Memory

- Stack is nice for intermediate values, but not for constants or lookup tables
- '*word*' = 32 bit, '*halfword*' = 16 bit, '*doubleword*' = 64 bit, '*byte*' = 8 bit, '*nibble*' = 4 bit
- Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- Ends up *somewhere* in RAM, need a label to access it
- For *n* bytes of uninitialized memory, use a label and `.skip n`
  - For *n* bytes of 0-initialized data, use `.lcomm somelabel, n`
- For global constants in ROM/flash, use `.section .rodata`

Memory

- Stack is nice for intermediate values, but not for constants or lookup tables
- '*word*' = 32 bit, '*halfword*' = 16 bit, '*doubleword*' = 64 bit, '*byte*' = 8 bit, '*nibble*' = 4 bit
- Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- Ends up *somewhere* in RAM, need a label to access it
- For *n* bytes of uninitialized memory, use a label and `.skip n`
  - For *n* bytes of 0-initialized data, use `.lcomm somelabel, n`
- For global constants in ROM/flash, use `.section .rodata`

Memory

- Stack is nice for intermediate values, but not for constants or lookup tables
- '*word*' = 32 bit, '*halfword*' = 16 bit, '*doubleword*' = 64 bit, '*byte*' = 8 bit, '*nibble*' = 4 bit
- Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- Ends up *somewhere* in RAM, need a label to access it
- For *n* bytes of uninitialized memory, use a label and `.skip n`
  - For *n* bytes of 0-initialized data, use `.lcomm somelabel, n`
- For global constants in ROM/flash, use `.section .rodata`

Using memory: `ldr/str`

- `adr r0, somelabel` to get the address in a register
- `ldr/str r1, [r0]` loads/stores a value
- `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- `ldr r1, [r0, r2, lsl #2]` is possible
  - if `r2` was a byte-offset, it's now used as word-offset
- `str` also has these variants

Using memory: `ldr/str`

- `adr r0, somelabel` to get the address in a register
- `ldr/str r1, [r0]` loads/stores a value
- `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- `ldr r1, [r0, r2, lsl #2]` is possible
  - if `r2` was a byte-offset, it's now used as word-offset
- `str` also has these variants

Using memory: `ldr/str`

- `adr r0, somelabel` to get the address in a register
- `ldr/str r1, [r0]` loads/stores a value
- `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- `ldr r1, [r0, r2, lsl #2]` is possible
  - if `r2` was a byte-offset, it's now used as word-offset
- `str` also has these variants

Using memory: `ldr/str`

- `adr r0, somelabel` to get the address in a register
- `ldr/str r1, [r0]` loads/stores a value
- `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- `ldr r1, [r0, r2, lsl #2]` is possible
  - if `r2` was a byte-offset, it's now used as word-offset
- `str` also has these variants

Using memory: `ldr/str`

- `adr r0, somelabel` to get the address in a register
- `ldr/str r1, [r0]` loads/stores a value
- `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- `ldr r1, [r0, r2, lsl #2]` is possible
  - if `r2` was a byte-offset, it's now used as word-offset
- `str` also has these variants

Using memory: `ldr/str`

- `adr r0, somelabel` to get the address in a register
- `ldr/str r1, [r0]` loads/stores a value
- `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- `ldr r1, [r0, r2, lsl #2]` is possible
  - if `r2` was a byte-offset, it's now used as word-offset
- `str` also has these variants

Using memory: `ldrd/strd/ldm/stm`

- `ldrd/strd r0, r1, [r2]` loads/stores two consecutive words from `r2`
  - Also as `ldrd/strd r0, r1, [r2, #4]` and `ldrd/strd r0, r1, [r2], #4`
- `ldm/stm r0, {r1,r2,r5}` loads/stores multiple from consecutive memory locations
- `ldm/stm r0!, {r1,r2,r5}` [...] and increments r0
- `push {r0,r1}` == `stmdb sp!, {r0,r1}`
  - *'store multiple decrement before'*
- **Caution:** `ldrd/strd/ldm/stm` require the address to be aligned to 4 bytes

Using memory: `ldrd/strd/ldm/stm`

- `ldrd/strd r0, r1, [r2]` loads/stores two consecutive words from `r2`
  - Also as `ldrd/strd r0, r1, [r2, #4]` and `ldrd/strd r0, r1, [r2], #4`
- `ldm/stm r0, {r1,r2,r5}` loads/stores multiple from consecutive memory locations
- `ldm/stm r0!, {r1,r2,r5}` [...] and increments r0
- `push {r0,r1}` == `stmdb sp!, {r0,r1}`
  - *'store multiple decrement before'*
- **Caution:** `ldrd/strd/ldm/stm` require the address to be aligned to 4 bytes

Using memory: `ldrd/strd/ldm/stm`

- `ldrd/strd r0, r1, [r2]` loads/stores two consecutive words from `r2`
  - Also as `ldrd/strd r0, r1, [r2, #4]` and `ldrd/strd r0, r1, [r2], #4`
- `ldm/stm r0, {r1,r2,r5}` loads/stores multiple from consecutive memory locations
- `ldm/stm r0!, {r1,r2,r5}` [...] and increments r0
- `push {r0,r1}` == `stmdb sp!, {r0,r1}`
  - *'store multiple decrement before'*
- **Caution:** `ldrd/strd/ldm/stm` require the address to be aligned to 4 bytes

Using memory: `ldrd/strd/ldm/stm`

- `ldrd/strd r0, r1, [r2]` loads/stores two consecutive words from `r2`
  - Also as `ldrd/strd r0, r1, [r2, #4]` and `ldrd/strd r0, r1, [r2], #4`
- `ldm/stm r0, {r1,r2,r5}` loads/stores multiple from consecutive memory locations
- `ldm/stm r0!, {r1,r2,r5}` [...] and increments r0
- `push {r0,r1}` == `stmdb sp!, {r0,r1}`
  - *'store multiple decrement before'*
- **Caution:** `ldrd/strd/ldm/stm` require the address to be aligned to 4 bytes

Using memory: Pipelining

- A single `ldr` instruction take 2 cycles (when not stalled)
- Two consecutive `ldr` instructions take 3 cycles
- N consecutive `ldr` instructions take $N + 1$ cycles
- `str` usually takes one cycle; does not pipeline
- `ldrd/strd/ldm/stm` do not pipeline together
  - `ldrd/strd` take 3 cycles
  - `ldm/stm` take $N + 1$ cycles
- There is some penalty for unaligned addresses for `ldr/str`
  - This may depend on your actual M4 core
- For more details look at `https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Load-store-timings`

Using memory: Pipelining

- A single `ldr` instruction take 2 cycles (when not stalled)
- Two consecutive `ldr` instructions take 3 cycles
- N consecutive `ldr` instructions take $N + 1$ cycles
- `str` usually takes one cycle; does not pipeline
- `ldrd/strd/ldm/stm` do not pipeline together
  - `ldrd/strd` take 3 cycles
  - `ldm/stm` take $N + 1$ cycles
- There is some penalty for unaligned addresses for `ldr/str`
  - This may depend on your actual M4 core
- For more details look at `https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Load-store-timings`

Using memory: Pipelining

- A single `ldr` instruction take 2 cycles (when not stalled)
- Two consecutive `ldr` instructions take 3 cycles
- N consecutive `ldr` instructions take $N + 1$ cycles
- `str` usually takes one cycle; does not pipeline
- `ldrd/strd/ldm/stm` do not pipeline together
  - `ldrd/strd` take 3 cycles
  - `ldm/stm` take $N + 1$ cycles
- There is some penalty for unaligned addresses for `ldr/str`
  - This may depend on your actual M4 core
- For more details look at `https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Load-store-timings`

Using memory: Pipelining

- A single `ldr` instruction take 2 cycles (when not stalled)
- Two consecutive `ldr` instructions take 3 cycles
- N consecutive `ldr` instructions take $N + 1$ cycles
- `str` usually takes one cycle; does not pipeline
- `ldrd/strd/ldm/stm` do not pipeline together
  - `ldrd/strd` take 3 cycles
  - `ldm/stm` take $N + 1$ cycles
- There is some penalty for unaligned addresses for `ldr/str`
  - This may depend on your actual M4 core
- For more details look at `https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Load-store-timings`

Subroutines

- `lr` keeps track of 'return address'
- Branch with link (`bl`) automatically sets `lr`
- Some performance overhead due to branching

```
somelabel:
    add r0, r1
    add r0, r1, ror #2
    add r0, r1, ror #4
    bx lr

main:
    bl somelabel
    mov r4, r0
    mov r0, r2
    mov r1, r3
    bl somelabel
    ...
```

Subroutines

- `lr` keeps track of 'return address'
- Branch with link (`bl`) automatically sets `lr`
- Some performance overhead due to branching

```
somelabel:
    add r0, r1
    add r0, r1, ror #2
    add r0, r1, ror #4
    bx lr

main:
    bl somelabel
    mov r4, r0
    mov r0, r2
    mov r1, r3
    bl somelabel
    ...
```

Application Binary Interface (ABI)

- Agreement on how to deal with parameters and return values

- If it fits, parameters in `r0-r3`
- Otherwise, a part in `r0-r3` and the rest on the stack
- Return value in `r0`

- The callee(!) should preserve `r4-r11` if it overwrites the
- `r12` is a scratch register (no need to preserve)
- Important when calling your assembly from, e.g., C

- For *private* subroutines: can ignore this ABI

Application Binary Interface (ABI)

- Agreement on how to deal with parameters and return values
- If it fits, parameters in `r0-r3`
- Otherwise, a part in `r0-r3` and the rest on the stack
- Return value in `r0`
- The callee(!) should preserve `r4-r11` if it overwrites the
- `r12` is a scratch register (no need to preserve)
- Important when calling your assembly from, e.g., C
- For *private* subroutines: can ignore this ABI

Application Binary Interface (ABI)

- Agreement on how to deal with parameters and return values

- If it fits, parameters in `r0-r3`
- Otherwise, a part in `r0-r3` and the rest on the stack
- Return value in `r0`

- The callee(!) should preserve `r4-r11` if it overwrites the
- `r12` is a scratch register (no need to preserve)
- Important when calling your assembly from, e.g., C

- For *private* subroutines: can ignore this ABI

Application Binary Interface (ABI)

- Agreement on how to deal with parameters and return values

- If it fits, parameters in `r0-r3`
- Otherwise, a part in `r0-r3` and the rest on the stack
- Return value in `r0`

- The callee(!) should preserve `r4-r11` if it overwrites the
- `r12` is a scratch register (no need to preserve)
- Important when calling your assembly from, e.g., C

- For *private* subroutines: can ignore this ABI

Application Binary Interface (ABI)

- Agreement on how to deal with parameters and return values

- If it fits, parameters in `r0-r3`
- Otherwise, a part in `r0-r3` and the rest on the stack
- Return value in `r0`

- The callee(!) should preserve `r4-r11` if it overwrites the
- `r12` is a scratch register (no need to preserve)
- Important when calling your assembly from, e.g., C

- For *private* subroutines: can ignore this ABI

Application Binary Interface (ABI)

- Agreement on how to deal with parameters and return values

- If it fits, parameters in `r0–r3`
- Otherwise, a part in `r0–r3` and the rest on the stack
- Return value in `r0`

- The callee(!) should preserve `r4–r11` if it overwrites the
- `r12` is a scratch register (no need to preserve)
- Important when calling your assembly from, e.g., C

- For *private* subroutines: can ignore this ABI

Architecture Reference Manual

- Large PDF that includes all of this, and more
- Available online: `https://developer.arm.com/documentation/ddi0403/latest/`
- See Chapter A7 for instruction listings and descriptions

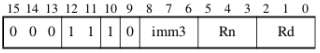
A6.7.3 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ADDS <Rd>,<Rn>,#<imm3>` Outside IT block.

`ADD<c> <Rd>,<Rn>,#<imm3>` Inside IT block.

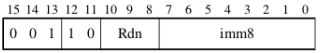



```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb ISA.

`ADDS <Rdn>,#<imm8>` Outside IT block.

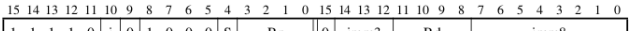
`ADD<c> <Rdn>,#<imm8>` Inside IT block.




```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

`ADD{S}<c>.W <Rd>,<Rn>,#<const>`



Architecture Reference Manual

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} <Rn>, #<const>
ADDW<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S`	If present, specifies that the instruction updates the flags. Otl update the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	Specifies the destination register. If `<Rd>` is omitted, this reg
`<Rn>`	Specifies the register that contains the first operand. If the S| *(SP plus immediate)* on page A6-26. If the PC is specified fc
`<const>`	Specifies the immediate value to be added to the value obta allowed values is 0-7 for encoding T1, 0-255 for encoding T See *Modified immediate constants in Thumb instructions* o allowed values for encoding T3.

Multiplications

- The M4 has numerous very powerful multiplication instructions
- They all take 1 cycle
- Most of them are only available in `Armv7E-M`, not `Armv7-M`

Multiplications (2)

Table A5-28 Multiply, multiply accumulate, and absolute difference operations

op1	op2	Ra	Instruction	See	Variant
000	00	not 1111	Multiply Accumulate	*MLA* on page A7-289	All
		1111	Multiply	*MUL* on page A7-302	All
	01	-	Multiply and Subtract	*MLS* on page A7-290	All
001	-	not 1111	Signed Multiply Accumulate, Halfwords	*SMLABB, SMLABT, SMLATB, SMLATT* on page A7-359	v7E-M
		1111	Signed Multiply, Halfwords	*SMULBB, SMULBT, SMULTB, SMULTT* on page A7-371	v7E-M
010	0x	not 1111	Signed Multiply Accumulate Dual	*SMLAD, SMLADX* on page A7-360	v7E-M
		1111	Signed Dual Multiply Add	*SMUAD, SMUADX* on page A7-370	v7E-M
011	0x	not 1111	Signed Multiply Accumulate, Word by halfword	*SMLAWB, SMLAWT* on page A7-364	v7E-M
		1111	Signed Multiply, Word by halfword	*SMULWB, SMULWT* on page A7-373	v7E-M
100	0x	not 1111	Signed Multiply Subtract Dual	*SMLSD, SMLSDX* on page A7-365	v7E-M
		1111	Signed Dual Multiply Subtract	*SMUSD, SMUSDX* on page A7-374	v7E-M
101	0x	not 1111	Signed Most Significant Word Multiply Accumulate	*SMMLA, SMMLAR* on page A7-367	v7E-M
		1111	Signed Most Significant Word Multiply	*SMMUL, SMMULR* on page A7-369	v7E-M
110	0x	-	Signed Most Significant Word Multiply Subtract	*SMMLS, SMMLSR* on page A7-368	v7E-M
111	00	1111	Unsigned Sum of Absolute Differences, Accumulate	*USADA8* on page A7-443	v7E-M
		not 1111	Unsigned Sum of Absolute Differences	*USAD8* on page A7-442	v7E-M

Multiplications (3)

Table A5-29 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-29 Long multiply, long multiply accumulate, and divide operations

op1	op2	Instruction	See	Variant
000	0000	Signed Multiply Long	*SMULL* on page A7-372	All
001	1111	Signed Divide	*SDIV* on page A7-350	All
010	0000	Unsigned Multiply Long	*UMULL* on page A7-435	All
011	1111	Unsigned Divide	*UDIV* on page A7-426	All
100	0000	Signed Multiply Accumulate Long	*SMLAL* on page A7-361	All
	10xx	Signed Multiply Accumulate Long, Halfwords	*SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page A7-362	v7E-M
	110x	Signed Multiply Accumulate Long Dual	*SMLALD, SMLALDX* on page A7-363	v7E-M
101	110x	Signed Multiply Subtract Long Dual	*SMLSLD, SMLSLDX* on page A7-366	v7E-M
110	0000	Unsigned Multiply Accumulate Long	*UMLAL* on page A7-434	All
	0110	Unsigned Multiply Accumulate Accumulate Long	*UMAAL* on page A7-433	v7E-M

Multiplications: `mul/mla/mls`

Table A4-4 General multiply instructions

Instruction	See	Operation (number of bits)
Multiply Accumulate	*MLA* on page A7-289	$32 = 32 + 32 \times 32$
Multiply and Subtract	*MLS* on page A7-290	$32 = 32 - 32 \times 32$
Multiply	*MUL* on page A7-302	$32 = 32 \times 32$

- `mul r0, r1, r2`
  - Computes `r1` · `r2` mod 2^{32} and writes it to `r0`
- `mla r0, r1, r2`
  - Computes `r1` · `r2` mod 2^{32} and adds it to `r0`
- `mls r0, r1, r2`
  - Computes `r1` · `r2` mod 2^{32} and subtracts it from `r0`
- As only the lower 32 bits are computed, there is no difference for signed/unsigned

Multiplications: `mul/mla/mls`

Table A4-4 General multiply instructions

Instruction	See	Operation (number of bits)
Multiply Accumulate	*MLA* on page A7-289	$32 = 32 + 32 \times 32$
Multiply and Subtract	*MLS* on page A7-290	$32 = 32 - 32 \times 32$
Multiply	*MUL* on page A7-302	$32 = 32 \times 32$

- `mul r0, r1, r2`
  - Computes `r1` $\cdot$ `r2` mod 2^{32} and writes it to `r0`
- `mla r0, r1, r2`
  - Computes `r1` $\cdot$ `r2` mod 2^{32} and adds it to `r0`
- `mls r0, r1, r2`
  - Computes `r1` $\cdot$ `r2` mod 2^{32} and subtracts it from `r0`
- As only the lower 32 bits are computed, there is no difference for signed/unsigned

Multiplications: `mul/mla/mls`

Table A4-4 General multiply instructions

Instruction	See	Operation (number of bits)
Multiply Accumulate	*MLA* on page A7-289	$32 = 32 + 32 \times 32$
Multiply and Subtract	*MLS* on page A7-290	$32 = 32 - 32 \times 32$
Multiply	*MUL* on page A7-302	$32 = 32 \times 32$

- `mul r0, r1, r2`
  - Computes `r1` $\cdot$ `r2` mod 2^{32} and writes it to `r0`
- `mla r0, r1, r2`
  - Computes `r1` $\cdot$ `r2` mod 2^{32} and adds it to `r0`
- `mls r0, r1, r2`
  - Computes `r1` $\cdot$ `r2` mod 2^{32} and subtracts it from `r0`
- As only the lower 32 bits are computed, there is no difference for signed/unsigned

Multiplications: `smull/smlal`

Table A4-5 Signed multiply instructions, Armv7-M base architecture

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate Long	*SMLAL* on page A7-361	64 = 64 + 32 × 32
Signed Multiply Long	*SMULL* on page A7-372	64 = 32 × 32

- `smull r0, r1, r2, r3`
  - Computes `r2` · `r3` and places the lower 32 bits in `r0` and the higher 32 bits in `r1`
  - `smull` for signed multiplication, `umull` for unsigned multiplication
- `smlal r0, r1, r2, r3`
  - Computes `r2` · `r3` and adds the 64-bit product to `r0, r1`
  - `smlal` for signed multiplication, `umlal` for unsigned multiplication

Multiplications: `smull/smlal`

Table A4-5 Signed multiply instructions, Armv7-M base architecture

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate Long	*SMLAL* on page A7-361	64 = 64 + 32 × 32
Signed Multiply Long	*SMULL* on page A7-372	64 = 32 × 32

- `smull r0, r1, r2, r3`
  - Computes `r2` · `r3` and places the lower 32 bits in `r0` and the higher 32 bits in `r1`
  - `smull` for signed multiplication, `umull` for unsigned multiplication
- `smlal r0, r1, r2, r3`
  - Computes `r2` · `r3` and adds the 64-bit product to `r0, r1`
  - `smlal` for signed multiplication, `umlal` for unsigned multiplication

Multiplications: More multiplications

Table A4-6 Signed multiply instructions, Armv7-M DSP extension (continued)

Instruction	See	Operation (number of bits)
Signed most significant Word Multiply	*SMMUL, SMMULR* on page A7-369	32 = 32 × 32[b]
Signed Dual Multiply Add	*SMUAD, SMUADX* on page A7-370	32 = 16 × 16 + 16 × 16
Signed Multiply, halfwords	*SMULBB, SMULBT, SMULTB, SMULTT* on page A7-371	32 = 16 × 16
Signed Multiply, word by halfword	*SMULWB, SMULWT* on page A7-373	32 = 32 × 16[a]
Signed Dual Multiply Subtract	*SMUSD, SMUSDX* on page A7-374	32 = 16 × 16 – 16 × 16

a. Uses the most significant 32 bits of the 48-bit product. Discards the less significant bits.
b. Uses the most significant 32 bits of the 64-bit product. Discards the less significant bits.

Multiplications: More multiplications (2)

- `smulbb r0, r1, r2`
  - Takes the lower 16 bits of `r1` and `r2`, computes 32-bit signed product
  - Similarly `smulbt`, `smultb`, `smultt` (`t` for upper 16 bits)
- `smuad r0, r1, r2`
  - Multiplies lower half of `r1` with lower half of `r2`
  - Multiplies upper half of `r1` with upper half of `r2`
  - Adds 32-bit products
  - `smuadx` computes `Lower(r1)·Upper(r2) + Upper(r1)·Lower(r2)`
- `smulwb r0, r1, r2`
  - Multiplies lower half of `r2` with full `r1` $\rightarrow$ 48-bit product
  - Writes upper 32 bit to `r0` (lower 16 bit discarded)
  - `smulwt` uses the upper half of `r2` instead
- ...

Multiplications: More multiplications (2)

- `smulbb r0, r1, r2`
  - Takes the lower 16 bits of `r1` and `r2`, computes 32-bit signed product
  - Similarly `smulbt`, `smultb`, `smultt` (`t` for upper 16 bits)
- `smuad r0, r1, r2`
  - Multiplies lower half of `r1` with lower half of `r2`
  - Multiplies upper half of `r1` with upper half of `r2`
  - Adds 32-bit products
  - `smuadx` computes `Lower(r1)·Upper(r2) + Upper(r1)·Lower(r2)`
- `smulwb r0, r1, r2`
  - Multiplies lower half of `r2` with full `r1` $\rightarrow$ 48-bit product
  - Writes upper 32 bit to `r0` (lower 16 bit discarded)
  - `smulwt` uses the upper half of `r2` instead
- ...

Multiplications: More multiplications (2)

- `smulbb r0, r1, r2`
  - Takes the lower 16 bits of `r1` and `r2`, computes 32-bit signed product
  - Similarly `smulbt`, `smultb`, `smultt` (`t` for upper 16 bits)
- `smuad r0, r1, r2`
  - Multiplies lower half of `r1` with lower half of `r2`
  - Multiplies upper half of `r1` with upper half of `r2`
  - Adds 32-bit products
  - `smuadx` computes `Lower(r1)·Upper(r2) + Upper(r1)·Lower(r2)`
- `smulwb r0, r1, r2`
  - Multiplies lower half of `r2` with full `r1` $\rightarrow$ 48-bit product
  - Writes upper 32 bit to `r0` (lower 16 bit discarded)
  - `smulwt` uses the upper half of `r2` instead
- ...

Multiplications: More multiplications (3)

Table A4-6 Signed multiply instructions, Armv7-M DSP extension

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate, halfwords	*SMLABB, SMLABT, SMLATB, SMLATT* on page A7-359	32 = 32 + 16 × 16
Signed Multiply Accumulate Dual	*SMLAD, SMLADX* on page A7-360	32 = 32 + 16 × 16 + 16 × 16
Signed Multiply Accumulate Long, halfwords	*SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page A7-362	64 = 64 + 16 × 16
Signed Multiply Accumulate Long Dual	*SMLALD, SMLALDX* on page A7-363	64 = 64 + 16 × 16 + 16 × 16
Signed Multiply Accumulate, word by halfword	*SMLAWB, SMLAWT* on page A7-364	32 = 32 + 32 × 16[a]
Signed Multiply Subtract Dual	*SMLSD, SMLSDX* on page A7-365	32 = 32 + 16 × 16 – 16 × 16
Signed Multiply Subtract Long Dual	*SMLSLD, SMLSLDX* on page A7-366	64 = 64 + 16 × 16 – 16 × 16
Signed most significant Word Multiply Accumulate	*SMMLA, SMMLAR* on page A7-367	32 = 32 + 32 × 32[b]
Signed most significant Word Multiply Subtract	*SMMLS, SMMLSR* on page A7-368	32 = 32 – 32 × 32[b]

Multiplications: More multiplications (4)

- `smlabb/smlabt/smlatb/smlatt`
  - Same as `smulbb/smulbt/smultb/smultt`, but with 32-bit accumulation
- `smlalbb/smlalbt/smlaltb/smlaltt`
  - Same as `smlabb/smulbt/smultb/smultt`, but with 64-bit accumulation
- `smlad/smladx`
  - Same as `smuad/smuadx`, but with 32-bit accumulation
- `smlawb/smlawt`
  - Same as `smulwb/smulwt`, but with 32-bit accumulation
- ...

Multiplications: More multiplications (4)

- `smlabb/smlabt/smlatb/smlatt`
  - Same as `smulbb/smulbt/smultb/smultt`, but with 32-bit accumulation
- `smlalbb/smlalbt/smlaltb/smlaltt`
  - Same as `smlabb/smulbt/smultb/smultt`, but with 64-bit accumulation
- `smlad/smladx`
  - Same as `smuad/smuadx`, but with 32-bit accumulation
- `smlawb/smlawt`
  - Same as `smulwb/smulwt`, but with 32-bit accumulation
- ...

Multiplications: More multiplications (4)

- `smlabb/smlabt/smlatb/smlatt`
  - Same as `smulbb/smulbt/smultb/smultt`, but with 32-bit accumulation
- `smlalbb/smlalbt/smlaltb/smlaltt`
  - Same as `smlabb/smulbt/smultb/smultt`, but with 64-bit accumulation
- `smlad/smladx`
  - Same as `smuad/smuadx`, but with 32-bit accumulation
- `smlawb/smlawt`
  - Same as `smulwb/smulwt`, but with 32-bit accumulation
- ...

Multiplications: More multiplications (4)

- `smlabb/smlabt/smlatb/smlatt`
  - Same as `smulbb/smulbt/smultb/smultt`, but with 32-bit accumulation
- `smlalbb/smlalbt/smlaltb/smlaltt`
  - Same as `smlabb/smulbt/smultb/smultt`, but with 64-bit accumulation
- `smlad/smladx`
  - Same as `smuad/smuadx`, but with 32-bit accumulation
- `smlawb/smlawt`
  - Same as `smulwb/smulwt`, but with 32-bit accumulation
- ...

Floating Point Unit — More Useful as Cache

Only Single Precision, Not Much Computation Power

A majority of ARM Cortex-M4 microcontrollers have a floating point unit (“M4F”).

- Handles 32-bit “single-precision” (6–7 significant digits) floating point numbers
- Can compute one multiply-accumulate in one cycle
- Affect its own flags, which must be moved into regular flags register for use
- Has 32-bit floating point registers (FPRs) `S0`, `S1`, …, `S31`.
  - FP registers `Sx` can serve as temporary storage for frequently used variables;
  - Moving data between General Purpose Registers (GPRs) and FPR’s is one cycle each with the `vmov` instruction
  - Loading data directly into FPRs from memory has the same latency as loading into GPRs, with the `vldr` and `vldm` instructions
  - Can put counters directly in FPRs

Floating Point Unit — More Useful as Cache

Only Single Precision, Not Much Computation Power

A majority of ARM Cortex-M4 microcontrollers have a floating point unit (“M4F”).

- Handles 32-bit “single-precision” (6–7 significant digits) floating point numbers
- Can compute one multiply-accumulate in one cycle
- Affect its own flags, which must be moved into regular flags register for use
- Has 32-bit floating point registers (FPRs) `S0`, `S1`, …, `S31`.
  - FP registers `Sx` can serve as temporary storage for frequently used variables;
  - Moving data between General Purpose Registers (GPRs) and FPR’s is one cycle each with the `vmov` instruction
  - Loading data directly into FPRs from memory has the same latency as loading into GPRs, with the `vldr` and `vldm` instructions
  - Can put counters directly in FPRs