# ARM

CSC 236

# ARM

- Advanced RISC Machine
  - Family of microprocessors
  - Designed by ARM holdings
  - Licenses semiconductor design
  - … but doesn’t manufacture any chips
- Advantages
  - Low transistor count
  - Low power usage
  - Ideal for embedded systems

# Dominates the embedded market

- 95% of phones
- 35% of hi-tech consumer devices
- Apple is using 8- to 10-core 5nm ARM processor in newest systems

Apple announced the M2 SoC on June 6, 2022, at WWDC, along with the new MacBook Air and the new 13-inch MacBook Pro and later the iPad Pro (6th generation). It is the successor to the Apple M1. The M2 is made with TSMC's "Enhanced 5-nanometer technology" N5P process and contains 20 billion transistors, a 25% increase from the previous generation M1. The M2 can be configured with up to 24 gigabytes of RAM and 2 terabytes of storage. It has 8 CPU cores (4 performance and 4 efficiency) and up to 10 GPU cores. The M2 also increases the memory bandwidth to 100 GB/s. Apple claims CPU improvements up to 18% and GPU improvements up to 35% compared to the previous M1

# Characteristics

- Load/store architecture
  - Data loaded from memory into registers
  - Operations on registers
  - Data stored from registers into memory
- Instructions
  - All are same size
  - Execute in one cycle*
- Two powerful extensions compared to 8086
  - Conditional execution
  - Optionally set condition codes
- Many variants
  - We'll look at the 32-bit architecture

**(*) per pipeline stage excluding load and store**

# Programmer's model

- 4GB address space (that's 32 bits of addresses)
- Data size
  - Byte (8 bits)
  - Halfword (16 bit)
  - Word (32 bits)
- Registers
  - 16 registers, each 32-bit
  - General purpose: r0 - r12
  - SP: r13 — stack pointer
  - LR: r14 — link register (like BP)
  - PC: r15 — program counter
  - A status register for condition codes

# Memory operations

- Load and store
  - Other instructions can’t access memory
  - Source is unmodified.
- Different instructions for:
  - Data size
  - Signed/unsigned

# Load

| Instruction | Operands | Notes |
|---|---|---|
| LDR | rd, =var | rd ⬅️ address of variable |
| LDR | rd, =const | rd ⬅️ immediate constant |
| LDR | rd, [rn] | rd ⬅️ value at rn |
| LDR | rd, [rn, rm] | rd ⬅️ value at rn + rm |
| LDR | rd, [rn, #n] | rd ⬅️ value at rn + #n |
| LDR | rd, [rn], #n | rd ⬅️ value at rn;<br>rn ⬅️ rn + #n |

I'm like a post-increment.

**No LDR r1, [var]**

**Instead:**

```
LDR r1, =var
LDR r1, [r1]
```

# Store

Notice, source and destination are reversed.

| Instruction | Operands | Notes |
|---|---|---|
| STR | rs, [rn] | [rn] ⬅️ rs |
| STR | rs, [rn, rm] | [rn + rm] ⬅️ rs |
| STR | rs, [rn, #n] | [rn + #n] ⬅️ rs |
| STR | rs, [rn], #n | [rn] ⬅️ rs;<br>rn ⬅️ rn + #n |

# Data Size

| Instruction | Data size |
|---|---|
| LDR | Word size (32 bits) |
| LDRH | Halfword, load into low-order 16 bits, zero-fill high-order 16 bits |
| LDRB | Byte, load into low-order 8 bits, zero-fill high-order 24 bits |
| LDRSH | Signed halfword, load into low-order 16 bits and sign-extend to 32 bits |
| LDRSB | Signed byte, load into low-order 8 bits and sign-extend to 32 bits |
| STR | Word size (32 bits) |
| STRH | Halfword, from the low-order 16 bits. |
| STRB | Byte, from the low-order 8 bits. |

Don't need signed versions of these.

# Move

- Load and store are for memory operations
- Not for copying between registers.

| Instruction | Operands | Notes |
|---|---|---|
| MOV | rd, rn | rd ⬅️ rn |

# Add and subtract

| Instruction | Operands | Notes |
|---|---|---|
| ADD | rd, rn, rm | rd ⬅️ rn + rm |
| ADD | rd, rn, #n | rd ⬅️ rn + #n |
| | | |
| SUB | rd, rn, rm | rd ⬅️ rn - rm |
| SUB | rd, rn, #n | rd ⬅️ rn - #n |

- Optionally set condition code

```
add r3,r2,r1 ;r3 = r2+r1
             ;CC not set

adds r3,r2,r1     ;r3 = r2+r1
             ;CC is set
```

**suffix "s" says "set condition codes"**
**This applies to many instructions**

# Multiply and Divide

| Instruction | Operands | Notes |
|---|---|---|
| UMULL | rdhi, rdlo, rm, rs | rdhi:rdlo ⬅️ rm * rs |
| SMULL | rdhi, rdlo, rm, rs | rdhi:rdlo ⬅️ rm * rs |
| | | |
| UDIV | rd, rm, rn | rd ⬅️ rn / rm |
| SDIV | rd, rm, rn | rd ⬅️ rn / rm |

Signed and Unsigned versions of these operations.

Multiply yields 64 bits

Divide yields 32 bits.

# Multiply and Divide

| Instruction | Operands | Notes |
|---|---|---|
| UMULL | rdhi, rdlo, rm, rs | rdhi:rdlo ⬅️ rm * rs |
| SMULL | rdhi, rdlo, rm, rs | rdhi:rdlo ⬅️ rm * rs |
| | | |
| UDIV | rd, rm, rn | rd ⬅️ rn / rm |
| SDIV | rd, rm, rn | rd ⬅️ rn / rm |

Some restrictions on what registers you can use here.

Divide may not be implemented in hardware.

# Compare

| Instruction | Operands | Notes |
|---|---|---|
| CMP | rd, rn | rd ? rn |
| CMP | rd, #n | rd ? #n |

- Like x86
  - Performs subtraction
  - Discards result
- Always sets the condition codes

# Status register

- 4 primary bits
  - Similar to Intel
  - N, Z, C, V — like SF, ZF, CF, OF
- N — negative
  - 0 ⇒ result is positive (or zero)
  - 1 ⇒ result is negative
  - Same as Intel SF
- Z — zero
  - 0 ⇒ result is not zero
  - 1 ⇒ result is zero
  - Same as Intel ZF
- V — signed overflow
  - 1 ⇒ signed overflow
  - 0 ⇒ no signed overflow
  - Same as Intel OF
- C — unsigned overflow
  - C = carry out
  - **Not exactly the same** as Intel CF

# Direct subtraction implementation

```
  0001
- 0010
```

# Direct subtraction implementation

```
  0001
- 0010
     1
```

# Direct subtraction implementation

```
  0001
- 0010
     1
```

# Direct subtraction implementation

**Assume leading 1**

```
 10001
-  0010
      1
```

# Direct subtraction implementation

**Redistribute**

```
   1
 0111
 ~~1~~0001
-  0010
      1
```

# Direct subtraction implementation

**Redistribute**

```
  1
 0111
 ~~1~~0001
-  0010
   1111
```

```
SF = 1
ZF = 0
OF =
CF =
```

# Direct subtraction implementation

```
    1
 0111
 ~~1~~0001
-  0010
   1111
```

```
SF  =  1
ZF  =  0
OF  =
CF  =
```

+1 - 2 — no signed overflow

# Direct subtraction implementation

```
   1
 0111
 +0001
-  0010
   1111
```

```
SF = 1
ZF = 0
OF = 0
CF =
```

# Direct subtraction implementation

```
    1
 0111
 ~~1~~0001
-  0010
   1111
```

```
SF = 1
ZF = 0
OF = 0
CF = 1
```

Did you need to borrow into the high-order bit.

Makes sense. 1 – 2 is an unsigned overflow.

# Status register

- 4 primary bits
  - Similar to Intel
  - N, Z, C, V — like SF, ZF, CF, OF
- N — negative
  - 0 ⇒ result is non-negative
  - 1 ⇒ result is negative
  - Same as Intel SF
- Z — zero
  - 0 ⇒ result is not zero
  - 1 ⇒ result is zero
  - Same as Intel ZF
- V — signed overflow
  - 1 ⇒ signed overflow
  - 0 ⇒ no signed overflow
  - Same as Intel OF
- C — unsigned overflow
  - C = carry out
  - **Not the same** as Intel CF
  - Semantics: depends on operation

| Unsigned overflow | Intel | ARM |
|---|---|---|
| Add | CF = 1 | C = 1 |
| Sub | CF = 1 | C = 0 |

# Subtraction on ARM

```
     1
  0111
 10001
-  0010
   1111
```

N = 1
Z = 0
V = 0
C = 0

What was left in that extra bit after subtraction?

```
     1
   0001
  10010
-  0001
   0001
```

N = 0
Z = 0
V = 0
C = 1

OK. It was still 1 after subtraction.

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| SIGNED | | | | |
| PL | Plus | N=0 | Positive result | |
| MI | Minus | N=1 | Negative result | |
| VC | Overflow clear | V=0 | No signed ovfl | |
| VS | Overflow set | V=1 | signed overflow | |
| GE | Greater / equal | N=V | result ≥ 0 | dest ≥ source |
| GT | Greater | Z=0 & N=V | result > 0 | dest > source |
| LE | Less / equal | Z=1 or N ≠ V | result ≤ 0 | dest ≤ source |
| LT | Less | N ≠ V | result < 0 | dest < source |

dest - source

These are the names for these comparisons.

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| SIGNED | | | | |
| PL | Plus | N=0 | Positive result | |
| MI | Minus | N=1 | Negative result | |
| VC | Overflow clear | V=0 | No signed ovfl | |
| VS | Overflow set | V=1 | signed overflow | |
| GE | Greater / equal | N=V | result ≥ 0 | dest ≥ source |
| GT | Greater | Z=0 & N=V | result > 0 | dest > source |
| LE | Less / equal | Z=1 or N ≠ V | result ≤ 0 | dest ≤ source |
| LT | Less | N ≠ V | result < 0 | dest < source |

dest - source

Result should be non-negative .. unless there's overflow

Same as 8086

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| SIGNED | | | | |
| PL | Plus | N=0 | Positive result | |
| MI | Minus | N=1 | Negative result | |
| VC | Overflow clear | V=0 | No signed ovfl | |
| VS | Overflow set | V=1 | signed overflow | |
| GE | Greater / equal | N=V | result ≥ 0 | dest ≥ source |
| GT | Greater | Z=0 & N=V | result > 0 | dest > source |
| LE | Less / equal | Z=1 or N ≠ V | result ≤ 0 | dest ≤ source |
| LT | Less | N ≠ V | result < 0 | dest < source |

dest - source

Negation of GE Condition

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| SIGNED | | | | |
| PL | Plus | N=0 | Positive result | |
| MI | Minus | N=1 | Negative result | |
| VC | Overflow clear | V=0 | No signed ovfl | |
| VS | Overflow set | V=1 | signed overflow | |
| GE | Greater / equal | N=V | result ≥ 0 | dest ≥ source |
| GT | Greater | Z=0 & N=V | result > 0 | dest > source |
| LE | Less / equal | Z=1 or N ≠ V | result ≤ 0 | dest ≤ source |
| LT | Less | N ≠ V | result < 0 | dest < source |

dest - source

GE Test, but excluding zero.

Negation of GT test.

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| SIGNED | | | | |
| PL | Plus | N=0 | Positive result | |
| MI | Minus | N=1 | Negative result | |
| VC | Overflow clear | V=0 | No signed ovfl | |
| VS | Overflow set | V=1 | signed overflow | |
| GE | Greater / equal | N=V | result ≥ 0 | dest ≥ source |
| GT | Greater | Z=0 & N=V | result > 0 | dest > source |
| LE | Less / equal | Z=1 or N ≠ V | result ≤ 0 | dest ≤ source |
| LT | Less | N ≠ V | result < 0 | dest < source |

Or, after an arithmetic operation

The condition codes tell you what you got.

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| UNSIGNED | | | | |
| CS | Carry set | C=1 | Add overflow | dest ≥ source |
| CC | Carry clear | C=0 | Sub overflow | dest < source |
| HI | High | C=1 & Z=0 | Sub result > 0 | dest > source |
| LS | Low or same | C=0 or Z=1 | Sub result ≤ 0 | dest ≤ source |
| BOTH SIGNED AND UNSIGNED | | | | |
| EQ | Equal | Z=1 | Equal | dest = source |
| NE | Not equal | Z=0 | Not equal | dest ≠ source |
| AL | Always | | Unconditional | |

dest - source

Remember, carry is backward on subtraction

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| UNSIGNED | | | | |
| CS | Carry set | C=1 | Add overflow | dest ≥ source |
| CC | Carry clear | C=0 | Sub overflow | dest < source |
| HI | High | C=1 & Z=0 | Sub result > 0 | dest > source |
| LS | Low or same | C=0 or Z=1 | Sub result ≤ 0 | dest ≤ source |
| BOTH SIGNED AND UNSIGNED | | | | |
| EQ | Equal | Z=1 | Equal | dest = source |
| NE | Not equal | Z=0 | Not equal | dest ≠ source |
| AL | Always | | Unconditional | |

dest - source

Negation of the CS (≥) test.

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| UNSIGNED | | | | |
| CS | Carry set | C=1 | Add overflow | dest ≥ source |
| CC | Carry clear | C=0 | Sub overflow | dest < source |
| HI | High | C=1 & Z=0 | Sub result > 0 | dest > source |
| LS | Low or same | C=0 or Z=1 | Sub result ≤ 0 | dest ≤ source |
| BOTH SIGNED AND UNSIGNED | | | | |
| EQ | Equal | Z=1 | Equal | dest = source |
| NE | Not equal | Z=0 | Not equal | dest ≠ source |
| AL | Always | | Unconditional | |

dest - source

CS test (≥), excluding zero.

Negation of HI (>) test.

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| UNSIGNED | | | | |
| CS | Carry set | C=1 | Add overflow | dest ≥ source |
| CC | Carry clear | C=0 | Sub overflow | dest < source |
| HI | High | C=1 & Z=0 | Sub result > 0 | dest > source |
| LS | Low or same | C=0 or Z=1 | Sub result ≤ 0 | dest ≤ source |
| BOTH SIGNED AND UNSIGNED | | | | |
| EQ | Equal | Z=1 | Equal | dest = source |
| NE | Not equal | Z=0 | Not equal | dest ≠ source |
| AL | Always | | Unconditional | |

Or, after an unsigned arithmetic operation.

This tells you what you got.

| | | Flags | Arithmetic | Compare |
|---|---|---|---|---|
| UNSIGNED | | | | |
| CS | Carry set | C=1 | Add overflow | dest ≥ source |
| CC | Carry clear | C=0 | Sub overflow | dest < source |
| HI | High | C=1 & Z=0 | Sub result > 0 | dest > source |
| LS | Low or same | C=0 or Z=1 | Sub result ≤ 0 | dest ≤ source |
| BOTH SIGNED AND UNSIGNED | | | | |
| EQ | Equal | Z=1 | Equal | dest = source |
| NE | Not equal | Z=0 | Not equal | dest ≠ source |
| AL | Always | | Unconditional | |

These tests make sense for signed or unsigned.

# Branch Instructions

| Instruction | Operands | Notes |
|---|---|---|
| beq | label | Branch if equal (Z=1) |
| bne | label | Branch if not equal (Z=0) |
| bcs | label | Branch if carry set (C=1, overflow on add, no overflow on subtract, dest ≥ source) |
| bpl | label | Branch if result is positive (N=0) |
| bge | label | Branch if greater than (Z=0 and N=V) |
| bal | label | Always jump |

We can use labels, just like with MASM.

A branch instruction for every condition, including “Always”

# Conditional execution

- Goal: abs(r4)

# Conditional execution

- Goal: abs(r4)
  - Depends on r0 = 0
  - (`ldr r0, #0`)

# Conditional execution

- Goal: abs(r4)
  - Depends on r0 = 0
  - (`ldr r0, #0`)
- Code

```
cmp r4, #0
```

# Conditional execution

- Goal: abs(r4)
  - Depends on r0 = 0
  - (`ldr r0, #0`)
- Code

```
cmp r4, #0
bpl skip
```

Branch if plus (result > 0)

# Conditional execution

- Goal: abs(r4)
  - Depends on r0 = 0
  - (`ldr r0, #0`)
- Code

```
   cmp r4, #0
   bpl skip
   sub r4,r0,r4
skip:
```

# Conditional execution

- Goal: abs(r4)
  - Depends on r0 = 0
  - (`ldr r0, #0`)
- Code

```
   cmp r4, #0
   bpl skip
   sub r4,r0,r4
skip:
```

- Condition execution
  - Execute sub conditionally
  - If result < 0
- Code

```
cmp r4, #0
~~bpl skip~~
submi r4,r0,r4
```

Execute sub if minus (result < 0)

# Conditional execution

- Goal: abs(r4)
  - Depends on r0 = 0
  - (`ldr r0, #0`)
- Code

```
   cmp r4, #0
   bpl skip
   sub r4,r0,r4
skip:
```

- Condition execution
  - Execute sub conditionally
  - If result < 0
- Code

```
cmp r4, #0
bpl skip
submi r4,r0,r4
```

These are the same names used by the conditionals.

# Conditional execution

- Fewer instructions
  - Yes!  That's good.
  - But the real reason for this is …
  - … pause for architecture lesson

# Pipelining

- Modern architectures are pipelined
  - Execution is divided into stages
  - Instructions pass from stage to stage
  - Like cars on an assembly line
  - Multiple instructions "in flight"
  - Increases parallelism
    - Working on multiple instructions at once
  - Increases clock speed
    - Less to do in each pipeline stage.

# Pipelining example

- Suppose there are 3 stages
  - Fetch
  - Decode
  - Execute
- Fetch
  - Gets instruction and operands (if any) from memory
- Decode
  - Executes logic to understand and dispatch instruction
- Execute
  - Performs requested operation

**Early ARM had 3-stage pipeline**

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | | | | | | | | |
| ADD | | | | | | | | |
| BEQ | | | | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

time

instructions

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | | | | | | | |
| ADD | | | | | | | | |
| BEQ | | | | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | | | | | | |
| ADD | | F | | | | | | |
| BEQ | | | | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| ADD | | F | D | | | | | |
| BEQ | | | F | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| ADD | | F | D | | | | | |
| BEQ | | | F | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

**What happens now?**
**Which instruction is next?**
**MOV?**
**CMP?**

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| ADD | | F | D | | | | | |
| BEQ | | | F | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

**ARM assumes branch NOT taken; fetches MOV instruction**

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| ADD | | F | D | E | | | | |
| BEQ | | | F | D | | | | |
| MOV | | | | F | | | | |
| STR | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| ADD | | F | D | E | | | | |
| BEQ | | | F | D | E | | | |
| MOV | | | | F | D | | | |
| STR | | | | | F | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

**Knows now if branch was taken**

**If not taken; keep going**

**What if branch is taken?**

# Pipelining example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| ADD | | F | D | E | | | | |
| BEQ | | | F | D | E | | | |
| MOV | | | | F | D | | | |
| STR | | | | | F | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | | | |

**Do not want these**

- **Flush pipeline**
- **Undo any effects**
- **No effects in this simple pipeline**

# Pipelining example

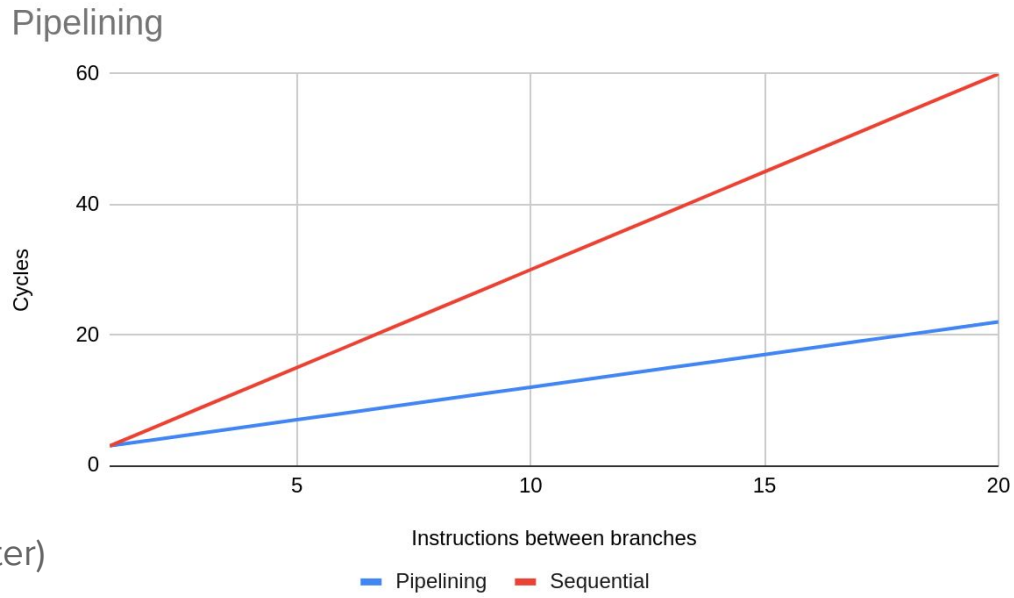
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| ADD | | F | D | E | | | | |
| BEQ | | | F | D | E | | | |
| MOV | | | | F | D | | | |
| STR | | | | | F | | | |
| | | | | | | | | |
| | | | | | | | | |
| CMP | | | | | | F | | |

**Do not want these**

- **Flush pipeline**
- **Undo any effects**
- **No effects in this simple pipeline**

# Pipelining performance

- One instruction
  - 3 cycles to complete
- N instructions
  - Without flushing pipeline
  - N + 2 cycles to complete
  - Not 3N
- Performance enhancement
  - Clock speed is ~ 3 times faster (shorter)
  - If average N instructions between branches (flushes)
  - N + 2 versus 3N



# Pipelining abs(r4)

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | | | | | | | | |
| CMP | | | | | | | | |
| SUBMI | | | | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| ... | | | | | | | | |
| ... | | | | | | | | |
| ... | | | | | | | | |

```
cmp r4, #0
submi r4,r0,r4
```

# Pipelining abs(r4)

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| CMP | | F | D | | | | | |
| SUBMI | | | F | | | | | |
| MOV | | | | | | | | |
| STR | | | | | | | | |
| ... | | | | | | | | |
| ... | | | | | | | | |
| ... | | | | | | | | |

# Pipelining abs(r4)

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| CMP | | F | D | E | | | | |
| SUBMI | | | F | D | E | | | |
| MOV | | | | F | D | | | |
| STR | | | | | F | | | |
| ... | | | | | | | | |
| ... | | | | | | | | |
| ... | | | | | | | | |

**Only execute if mi (minus)**

# Pipelining abs(r4)

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| LDR | F | D | E | | | | | |
| CMP | | F | D | E | | | | |
| SUBMI | | | F | D | E | | | |
| MOV | | | | F | D | E | | |
| STR | | | | | F | D | E | |
| ... | | | | | | F | D | E |
| ... | | | | | | | F | D |
| ... | | | | | | | | F |

# Conditional Execution

| | Conditionally executed | Optionally set CC |
|---|---|---|
| ADD /SUB | Yes | Yes — N,Z,C,V |
| AND / ORR / EOR | Yes | Yes — N,Z,C |
| BXX | Yes | No |
| CMP | Yes | Always |
| LDR / STR | Yes | No |
| MOV | Yes | Yes — N,Z,C |
| STMDB / LDMIA | Yes | No |
| UMUL / SMULL | Yes | Yes — N,Z |

- How to code
  - Conditional execution
  - Set CC
- Format

  opcodecondset

- cond — Execution depends on CC
  - addmi
  - subpl
- set — 's'
  - adds
  - subs

**Can be combined:**
addmis

# Immediate data

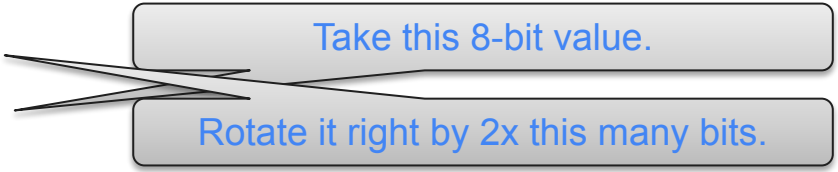
- `add r0, r0, #1` — r0=r0 +1
- `add r0, r0, #10` — r0=r0 +10
- `add r0, r0, #100` — r0=r0 +100
- `add r0, r0, #999` — illegal
- `add r0, r0, #1000` — r0=r0 +1000
- `add r0, r0, #1001` — illegal

# Immediate data

- X86 operands can be encoded in additional bytes.
  - If immediate is byte — data component is 1 byte
  - If immediate is word — data component is 2 bytes
  - Variable length instructions
- ARM machine instructions are always 32 bits
- Bits are needed for
  - Opcode
  - Operands
- We can’t spend all 32 bits on an immediate value … what to do?

# Immediate data

- ARM uses 12 bits for immediate value
  - Rotate — 4 bits 0-15
  - Value — 8 bits 0-255




- Why rotate **right**?
  - Rotate left by N bits is just rotate right by 32 – N.
- This produces a 32-bit value
  - But just some 32-bit values.
  - … hopefully, the ones you need most often.

# Immediate data

| **ROT** | F | E | D | C | ... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **2 x ROT** | 30 | 28 | 26 | 24 | | 2 | 0 |
| **Rot left** | 2 | 4 | 6 | 8 | | 30 | 0 |
| **Multiply** | 4 | 16 | 64 | 256 | | $2^{30}$ | 1 |

# Immediate data

| **ROT** | F | E | D | C | ... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **2 x ROT** | 30 | 28 | 26 | 24 | | 2 | 0 |
| **Rot left** | 2 | 4 | 6 | 8 | | 30 | 0 |
| **Multiply** | 4 | 16 | 64 | 256 | | $2^{30}$ | 1 |

`0000 0000 0000 0000 0000 0000 abcd efgh`

Any sequence of 8 bits.

# Immediate data

| **ROT** | F | E | D | C | ... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **2 x ROT** | 30 | 28 | 26 | 24 | | 2 | 0 |
| **Rot left** | 2 | 4 | 6 | 8 | | 30 | 0 |
| **Multiply** | 4 | 16 | 64 | 256 | | $2^{30}$ | 1 |

```
0000 0000 0000 0000 0000 0000 abcd efgh

0000 0000 0000 0000 0000 abcd efgh 0000
```

# Immediate data

| **ROT** | F | E | D | C | ... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **2 x ROT** | 30 | 28 | 26 | 24 | | 2 | 0 |
| **Rot left** | 2 | 4 | 6 | 8 | | 30 | 0 |
| **Multiply** | 4 | 16 | 64 | 256 | | $2^{30}$ | 1 |

```
0000 0000 0000 0000 0000 0000 abcd efgh

0000 0000 0000 0000 abcd efgh 0000 0000
```

# Immediate data

| **ROT** | F | E | D | C | ... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **2 x ROT** | 30 | 28 | 26 | 24 | | 2 | 0 |
| **Rot left** | 2 | 4 | 6 | 8 | | 30 | 0 |
| **Multiply** | 4 | 16 | 64 | 256 | | $2^{30}$ | 1 |

```
0000 0000 0000 0000 0000 0000 abcd efgh

gh00 0000 0000 0000 0000 0000 00ab cdef
```

# Immediate data

| **ROT** | F | E | D | C | ... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **2 x ROT** | 30 | 28 | 26 | 24 | | 2 | 0 |
| **Rot left** | 2 | 4 | 6 | 8 | | 30 | 0 |
| **Multiply** | 4 | 16 | 64 | 256 | | $2^{30}$ | 1 |

- **Generate #1000**
  - **1000 = $1111101000_2$ = $11111010_2$ * 4**
  - **Rotate = F**
  - **Value = $11111010_2$ = FA**

# Immediate data

| **ROT** | F | E | D | C | ... | 1 | 0 |
|---|---|---|---|---|---|---|---|
| **2 x ROT** | 30 | 28 | 26 | 24 | | 2 | 0 |
| **Rot left** | 2 | 4 | 6 | 8 | | 30 | 0 |
| **Multiply** | 4 | 16 | 64 | 256 | | $2^{30}$ | 1 |

- **Cannot generate #999** **(that's $1111100111_2$)**
- **Cannot generate #1001** **(that's $1111101001_2$)**

兩位 0