Key ARM ISA Features for Performance

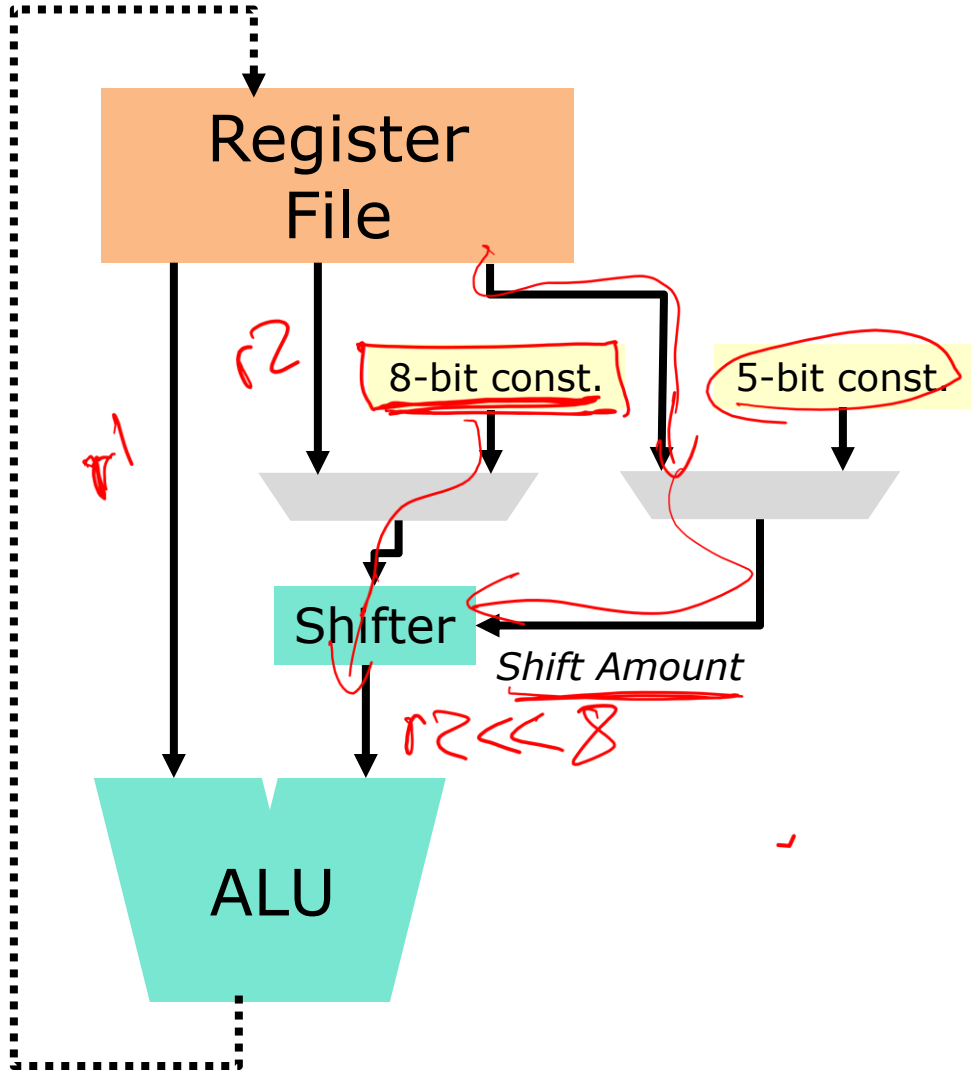
Key Features of ISA

[section references in Cortex A-Series Programmer's Guide]

- Barrel shifter for operand 2 [5.2.1]
  - Integrates power-of-two multiply/divide into single instruction
- Advanced addressing modes [5.3.1]
  - Integrate offsets and pointers updates into single instruction
- Load/Store multiple [5.3.2]
  - … with a single instruction (includes pointer updates)
- Conditional instruction execution [5.1.2]
  - Not just for branches: ADDNE vs. BNE
  - Eliminates conditional branches for short code segments
- Cache preload instructions [5.6.5]
  - PLD (data) and PLI (instruction)
- Table branches [A.1.154, 155]
  - Use PC-relative forward branching with table of offsets. Rn = table address, Rm = index

Barrel Shifter [5.2.1]

- Barrel shifter provided between register file and ALU
- Single cycle shift, rotate etc. regardless of shift amount
- If shift amount is in register, may take an extra cycle.
  - Extra credit to evaluate on Cortex-A72?
- Can also perform "free" shift on last operand of an instruction
  - 8-bit constant rotated right (ROR) through even number of positions
    - ADD r0, r1, #0xc5, 10
    - r0 <= r1 + (0xc5 rotated right 10 bits)
    - 0xc5 = 11000101
    - Rotate right 10 times (or rotate left by 22 bits)
    - 0011 0001 0100 0000 0000 0000 0000 0000
  - Register shifted (LSL, ASR) or rotated (ROR, RRX) by constant or another register
    - SUB r0, r1, r2, LSR #10
    - r0 <= r1 – (r2>>10)



Addressing Modes Leverage Barrel Shifter [5.3.1]

- Form address by adding offset to register
- Offset options
  - Unsigned 12 bit constant (allows offset of 0 to 4095)
  - Register contents shifted/rotated by a five-bit constant
- Examples
  - STR r7, [r0], #24
    - Post-indexed
    - Memory[r0] <= r7
    - r0 <= r0+24
  - LDR r2, [r0], r4, ASR #4
    - Post-indexed
    - r2 <= Memory[r0]
    - r0 <= r0 + r4>>4
  - STR r3, [r0, r5, LSL #3]
    - Pre-indexed
    - Memory[r0 + (r5<<3)] <= r3
  - LDR r6, [r0, r1, ROR#6]!
    - Pre-indexed, write back
    - r6 <= Memory[r0 + (r1>>6)]
    - r0 <= r0 + (r1>>6)

Load/Store Multiple Instructions [5.3.2]

- LDM/STM: move several registers to/from memory
  - Smaller code size, faster execution
  - Order in instruction does not matter: smallest register number stored at smallest address
- Addressing modes
  - IA: increment after, IB: increment before
  - DA: decrement after, DB: decrement before
- !: Base register writeback

- STMIA r10, {r1, r3-r5, r8}
  - Memory[r10] <= r1
  - Memory[r10+4] <= r3
  - Memory[r10+8] <= r4
  - Memory[r10+12] <= r5
  - Memory[r10+16] <= r8
  - Doesn't change r10
- LDMIB r11!, {r9, r4-r7}
  - r4 <= Memory[r11+4]
  - r5 <= Memory[r11+8]
  - r6 <= Memory[r11+12]
  - r7 <= Memory[r11+16]
  - r9 <= Memory[r11+20]
  - r11 <= r11 + 20

pushm

popm

Conditional Instruction Execution [5.1.2]

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
    CMP   r3,#0
    BEQ   skip
    ADD   r0,r1,r2
skip
```

```
    CMP   r3,#0
    ADDNE r0,r1,r2
```

ZNC
1 1 1 1
0

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

```
loop
    ...
    SUBS r1,r1,#1
    BNE  loop
```

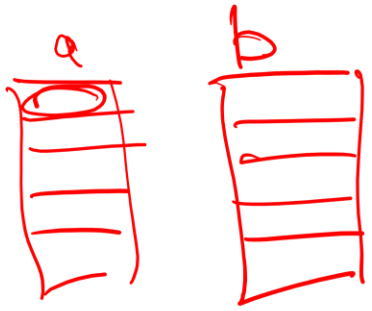
decrement r1 and set flags

if Z flag clear then branch

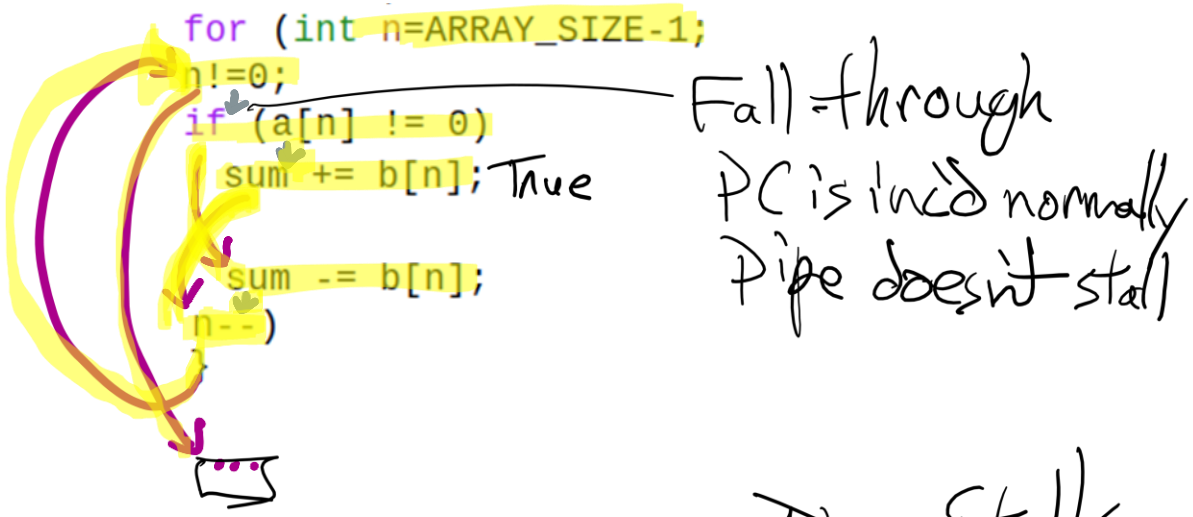
Example: Removing Control Flow Hazards

- Source code: Speed/Scalar/compiler/opts.c

```
int do_test1(void) {
  int sum=0;
  for (int n=ARRAY_SIZE-1; n!=0; n--) {
    if (a[n] != 0)
      sum += b[n];
    else
      sum -= b[n];
  }
  return sum;
}
```



- Consider code layout and changes in control flow (vs. fall-through)



- Branches are slow in a pipelined processor if mis-predicted
  - So want to avoid changing control flow in program

- Possible control-flow changes – Pipe Stalls
  - n != 0 Loop-test branch easy to predict
  - a[n] != 0 test is data-dependent, hard to predict
  - Exit if true case
  - Loop back edge

Example: Removing Control Flow Hazards

- Use driver program
  - Loads array a[] so branches flip back and forth under our control
  - 0, 1, 0, 1, 0, 1, 0, 1, etc.

```
int do_test1(void) {
  int sum=0;
  for (int n=ARRAY_SIZE-1; n!=0; n--) {
    if (a[n] != 0)
      sum += b[n];
    else
      sum -= b[n];
  }
  return sum;
}
```

- Build* and run
  - * *Start by compiling with –O3 –fno-if-conversion2 to show base case without ISA support for predication*

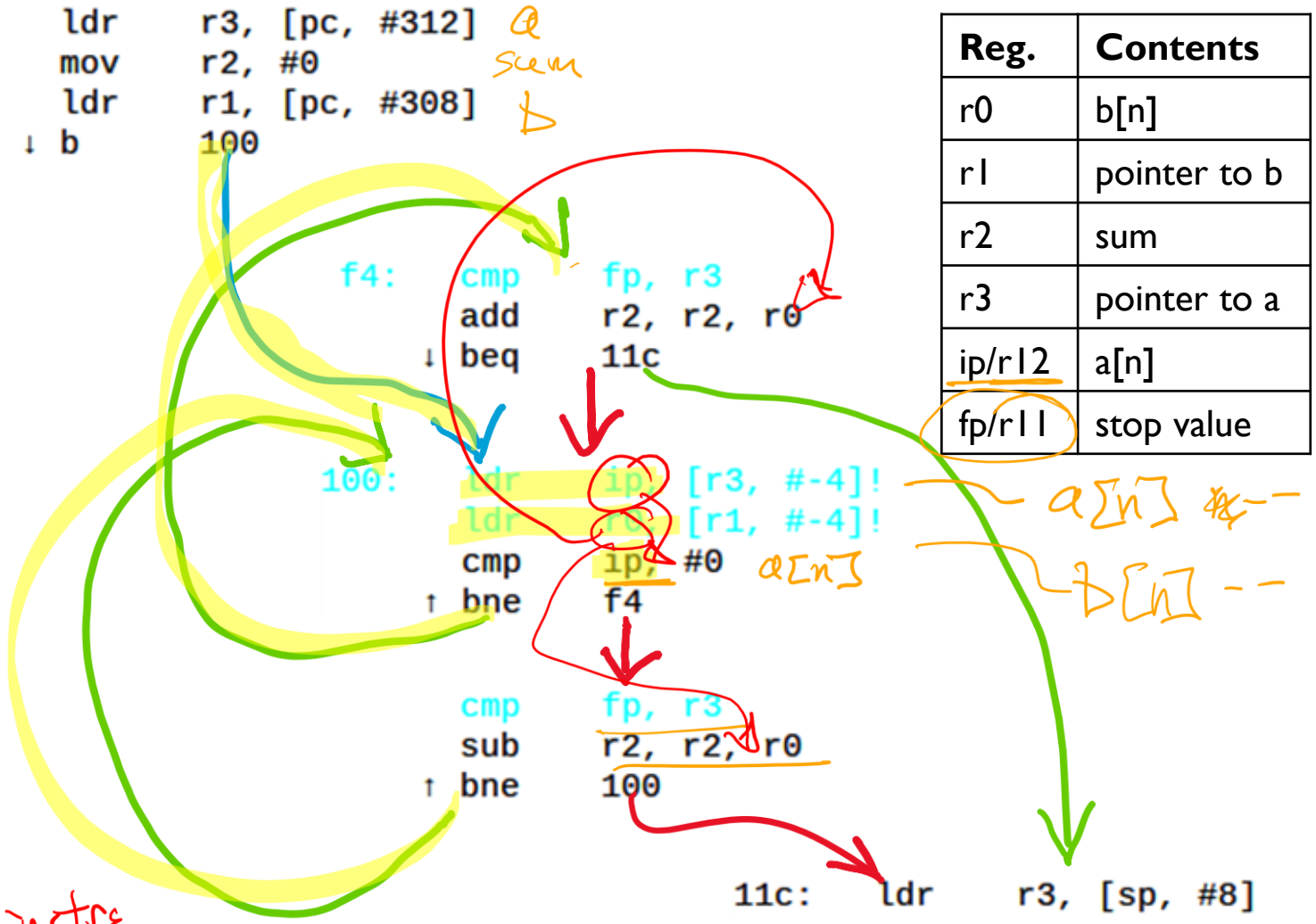
- Performance: 5.725 cycles per array element. Is this good or bad?
  - Check the object code!

```
pi@raspberrypi:~/AES-2020/Speed/Scalar/compiler $ ./opts
Assuming clock of 1.5 GHz
10000 elements in array, 10000 tests, mask length is 1 bits
Mask 0x1 leads to a[n]==0 every 2 entries]
Cycles per array element: minimum  5.708,    average  5.725
pi@raspberrypi:~/AES-2020/Speed/Scalar/compiler $
```

Examining the Object Code

```
int do_test1(void) {
  int sum=0;
  for (int n=ARRAY_SIZE-1; n!=0; n--) {
    if (a[n] != 0)
      sum += b[n];
    else
      sum -= b[n];
  }
  return sum;
}
```

- Look at all those branches!
- Only 7 instructions execute per loop iteration
  - Two are loads, two are conditional branches
  - So maybe this is not so bad? But non-load instructions are taking significant time…



Reg.	Contents
r0	b[n]
r1	pointer to b
r2	sum
r3	pointer to a
ip/r12	a[n]
fp/r11	stop value

Examining the Profile

```
int do_test1(void) {
  int sum=0;
  for (int n=ARRAY_SIZE-1; n!=0; n--) {
    if (a[n] != 0)
      sum += b[n];
    else
      sum -= b[n];
  }
  return sum;
}
```

Reg.	Contents
r0	b[n]
r1	pointer to b
r2	sum
r3	pointer to a
ip/r12	a[n]
fp/r11	stop value

```
  ldr   r3, [pc, #312]
  mov   r2, #0
  ldr   r1, [pc, #308]
↓ b     100
```

```
f4:   cmp   fp, r3
      add   r2, r2, r0
    ↓ beq   11c
100:  ldr   ip, [r3, #-4]!
      ldr   r0, [r1, #-4]!
      cmp   ip, #0
    ↑ bne   f4
      cmp   fp, r3
      sub   r2, r2, r0
    ↑ bne   100
11c:  ldr   r3, [sp, #8]
```

Control flow stall

```
0.40          ldr   r3, [pc, #312]
              mov   r2, #0
              ldr   r1, [pc, #308]
            ↓ b     100
9.29  f4:     cmp   fp, r3
              add   r2, r2, r0
            ↓ beq   11c
55.85 100:    ldr   ip, [r3, #-4]!
25.16         ldr   r0, [r1, #-4]!
              cmp   ip, #0
            ↑ bne   f4
 8.65         cmp   fp, r3
              sub   r2, r2, r0
            ↑ bne   100
0.16  11c:    ldr   r3, [sp, #8]
```

a[n]
b[n]

Good Enough? Do a Sensitivity Analysis

- Consider the branches
  - Loop-test branch should be easily predicted (always repeat except for iteration #10,000)
  - Data test branch (a[n]!=0) is more difficult
    - Testing the code with alternating taken/not taken branches: T N T N T N
    - Might really be stressing the branch predictor
- Use driver code lets us try different patterns based on mask length in bits
  - mask length 1: T N
  - mask length 2 : T T T N
  - mask length 3 : T T T T T T T N
- Branch predictor should benefit from longer runs (and longer masks)

```
pi@raspberrypi:~/AES-2020/Speed/Scalar/compiler $ ./opts 10
Assuming clock of 1.5 GHz
10000 elements in array, 10000 tests, mask length is 10 bits
Mask 0x3ff leads to a[n]==0 every 1024 entries]
Cycles per array element: minimum  2.800,    average  2.819
Mask 0x1ff leads to a[n]==0 every 512 entries]
Cycles per array element: minimum  2.819,    average  2.836
Mask 0xff leads to a[n]==0 every 256 entries]
Cycles per array element: minimum  2.858,    average  2.874
Mask 0x7f leads to a[n]==0 every 128 entries]
Cycles per array element: minimum  2.942,    average  2.955
Mask 0x3f leads to a[n]==0 every 64 entries]
Cycles per array element: minimum  3.100,    average  3.112
Mask 0x1f leads to a[n]==0 every 32 entries]
Cycles per array element: minimum  2.919,    average  3.437
Mask 0xf leads to a[n]==0 every 16 entries]
Cycles per array element: minimum  2.778,    average  2.794
Mask 0x7 leads to a[n]==0 every 8 entries]
Cycles per array element: minimum  3.214,    average  3.226
Mask 0x3 leads to a[n]==0 every 4 entries]
Cycles per array element: minimum  4.208,    average  4.220
Mask 0x1 leads to a[n]==0 every 2 entries]
Cycles per array element: minimum  5.708,    average  5.723
pi@raspberrypi:~/AES-2020/Speed/Scalar/compiler $
```

Cycles

Run Length

What to Do?

- Good thing we checked – double the performance by not mis-predicting branches?

- How? Use instruction set support for conditional execution (predication) of instructions
  - Instruction commits its results if its condition (in suffix) is true
  - Also called *if-conversion*: turns if/else structures into basic blocks without control flow
  - Converts control dependencies to data dependencies -- microarchitecture can resolve these quickly

How to Do It?

- GCC automatically enables if-conversion with –O, -O2, -O3, -Os
- Avoid using suspicious compiler optimization flag **–fno-if-conversion2** in Makefile (**no-** prefix disables optimization)

`-fif-conversion`

Attempt to transform conditional jumps into branch-less equivalents. This includes use of conditional moves, min, max, set flags and abs instructions, and some tricks doable by standard arithmetics. The use of conditional execution on chips where it is available is controlled by `-fif-conversion2`.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`, but not with `-Og`.

`-fif-conversion2`

Use conditional execution (where available) to transform conditional jumps into branch-less equivalents.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`, but not with `-Og`.

```
# No if-conversion
CFLAGS = -c -Wall  -ggdb -O3  -mfloat-abi=hard -mcpu=cortex-a72 -mfpu=crypto-neon-fp-armv8 -fno-tree-vectorize  -fno-if-conversion2
# if-conversion
# CFLAGS = -c -Wall  -ggdb -O3  -mfloat-abi=hard -mcpu=cortex-a72 -mfpu=crypto-neon-fp-armv8 -fno-tree-vectorize
```

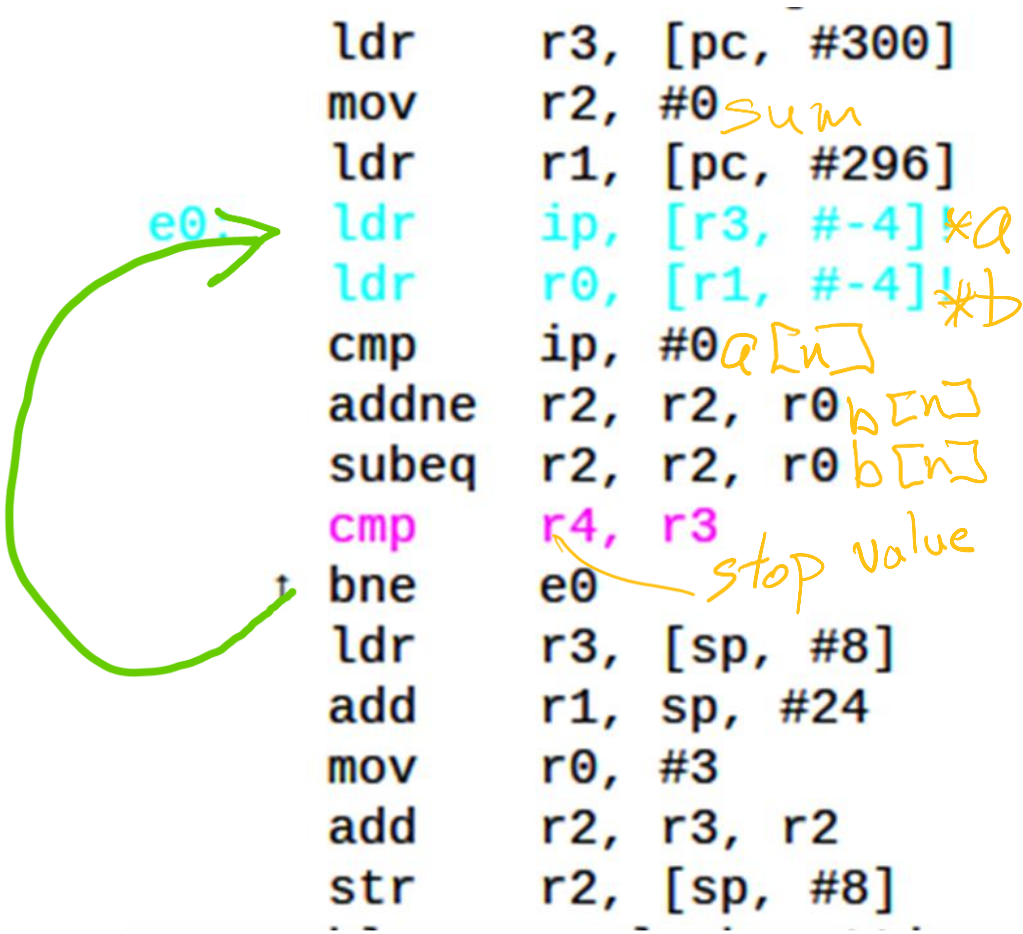
ISA Features for Removing Control Flow Hazards

```
int do_test1(void) {
  int sum=0;
  for (int n=ARRAY_SIZE-1; n!=0; n--) {
    if (a[n] != 0)
      sum += b[n];
    else
      sum -= b[n];
  }
  return sum;
}
```

Reg.	Contents
r0	b[n]
r1	pointer to b
r2	sum
r3	pointer to a
ip	a[n]
r4	stop value

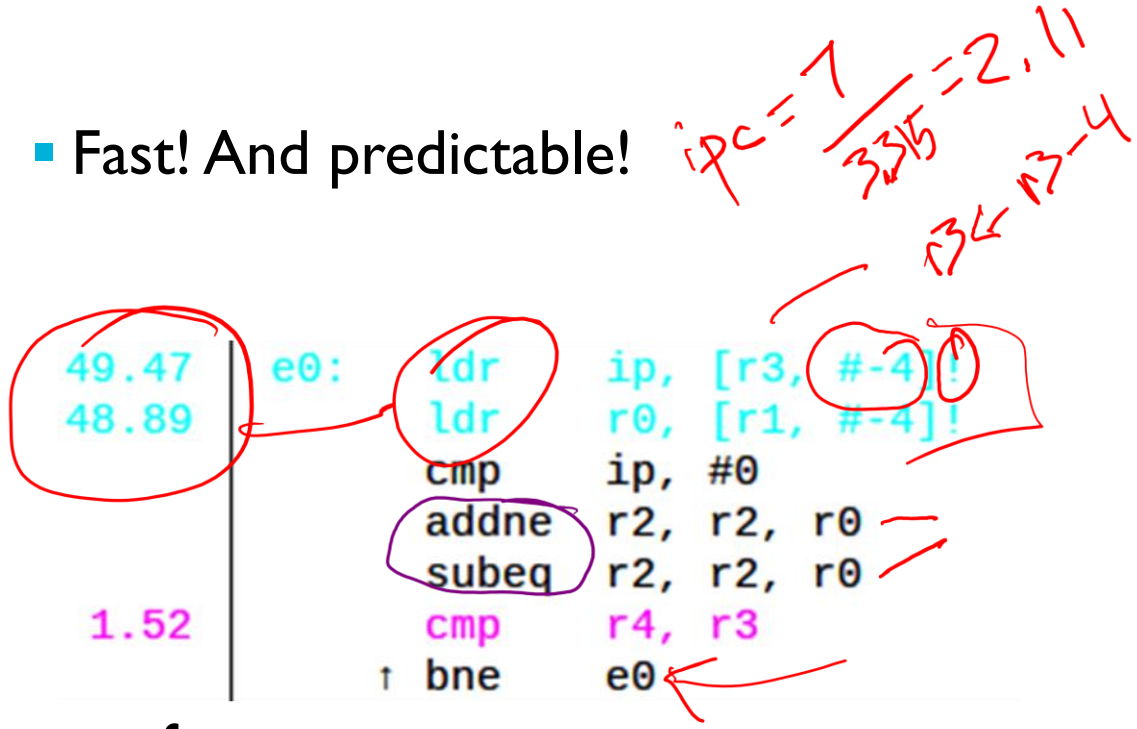
```
      ldr    r3, [pc, #300]
      mov    r2, #0
      ldr    r1, [pc, #296]

e0:   ldr    ip, [r3, #-4]!
      ldr    r0, [r1, #-4]!
      cmp    ip, #0
      addne  r2, r2, r0
      subeq  r2, r2, r0
      cmp    r4, r3
      bne    e0

      ldr    r3, [sp, #8]
```

- New object code looks great!
  - add**ne**
  - sub**eq**
  - Only one branch!
- But is it fast?



Performance of Code with If-Conversion

- Fast! And predictable!

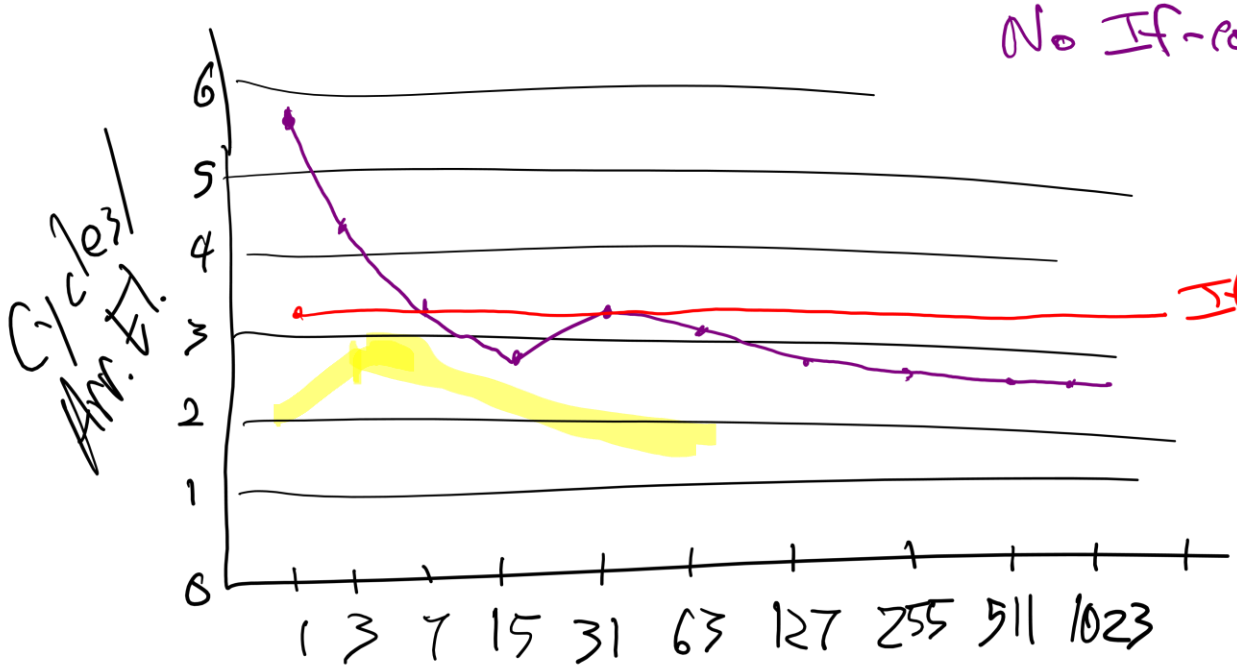


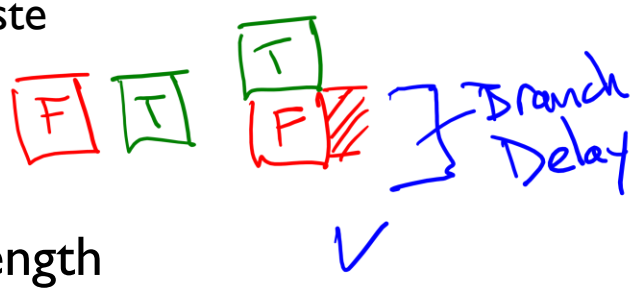
- perf report
- Notice how the load register instructions now dominate profile
  - Our next challenge?

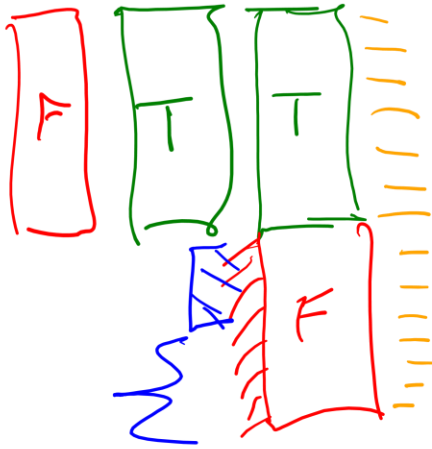
```
pi@raspberrypi:~/AES-2020/Speed/Scalar/compiler $ ./opts 10
Assuming clock of 1.5 GHz
10000 elements in array, 10000 tests, mask length is 10 bits
Mask 0x3ff leads to a[n]==0 every 1024 entries]
Cycles per array element: minimum  3.303,    average  3.318
Mask 0x1ff leads to a[n]==0 every 512 entries]
Cycles per array element: minimum  3.303,    average  3.315
Mask 0xff leads to a[n]==0 every 256 entries]
Cycles per array element: minimum  3.303,    average  3.315
Mask 0x7f leads to a[n]==0 every 128 entries]
Cycles per array element: minimum  3.303,    average  3.315
Mask 0x3f leads to a[n]==0 every 64 entries]
Cycles per array element: minimum  3.300,    average  3.314
Mask 0x1f leads to a[n]==0 every 32 entries]
Cycles per array element: minimum  3.300,    average  3.315
Mask 0xf leads to a[n]==0 every 16 entries]
Cycles per array element: minimum  3.303,    average  3.315
Mask 0x7 leads to a[n]==0 every 8 entries]
Cycles per array element: minimum  3.303,    average  3.315
Mask 0x3 leads to a[n]==0 every 4 entries]
Cycles per array element: minimum  3.300,    average  3.314
Mask 0x1 leads to a[n]==0 every 2 entries]
Cycles per array element: minimum  3.303,    average  3.315
pi@raspberrypi:~/AES-2020/Speed/Scalar/compiler $
```

Performance Comparison

- Limits to performance improvement
  - Longer runs are easier for branch predictor, so less benefit from eliminating the if/else branches (shown in graph)
  - Instructions from both cases use processor and consume resources (e.g. time). Waste time if a case is longer than delay for mis-predicted branch
- Why the bump at run length = 31?

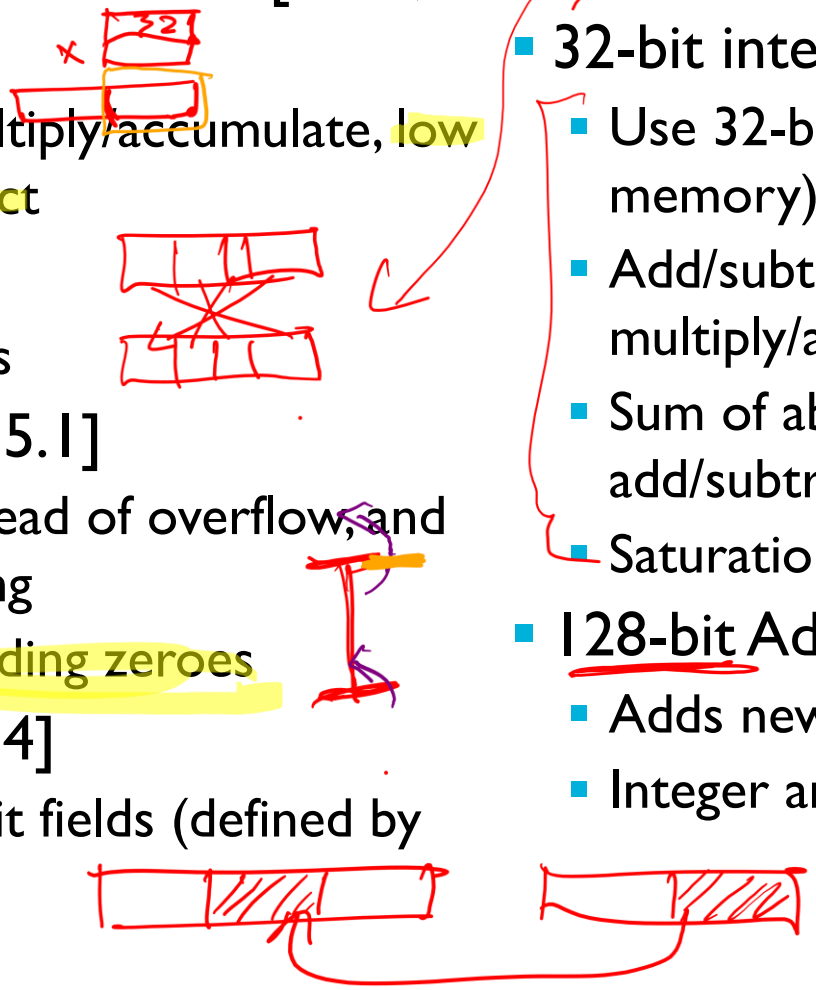




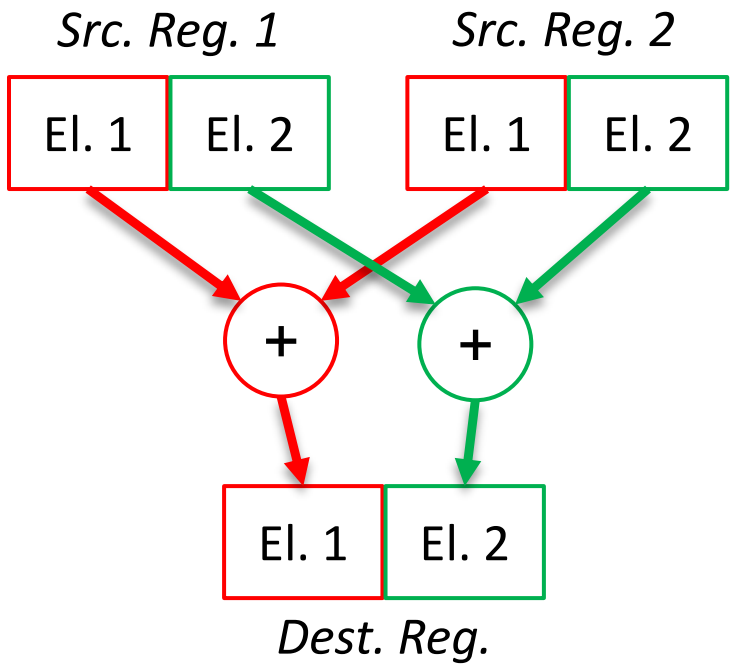


Key Features of ISA – Data Processing

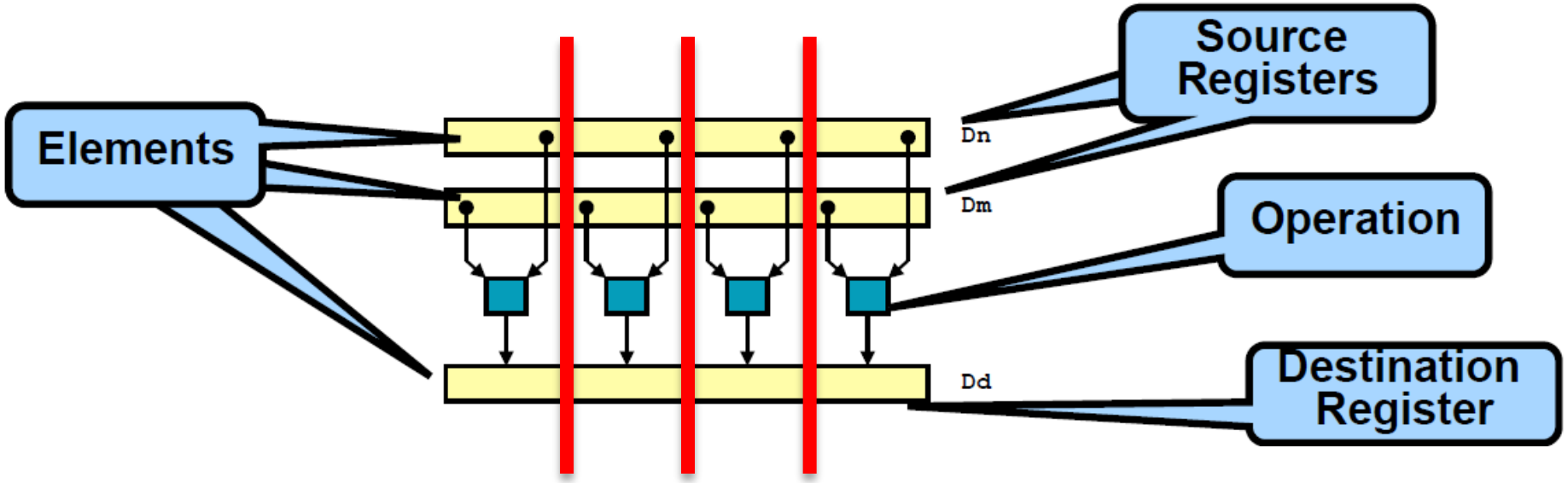
- Integer multiplication instructions [5.2.2, 5.2.3]
  - 32 and 64 bit results, multiply/accumulate, low and high words of product
- Byte reversal [5.6.6]
  - Swap bytes or half-words
- Saturating arithmetic [5.5.1]
  - Operations saturate instead of overflow, and set sticky Q bit for testing
  - Instructions to count leading zeroes
- Bit field operations [5.6.4]
  - Insert, clear or extract bit fields (defined by LSB position and width)
  - Reverse bits within register
- 32-bit integer SIMD instructions [5.2.4]
  - Use 32-bit data path (including registers and memory) as two half-words or four bytes
  - Add/subtract with exchange, multiply/accumulate,
  - Sum of absolute differences, multiply add/subtract products,
  - Saturation, packing, extraction, select bytes
- 128-bit Advanced SIMD instructions
  - Adds new 128-bit-wide register file
  - Integer and floating point operations



32-BIT SIMD



SIMD: Single Instruction performed on Multiple Data items

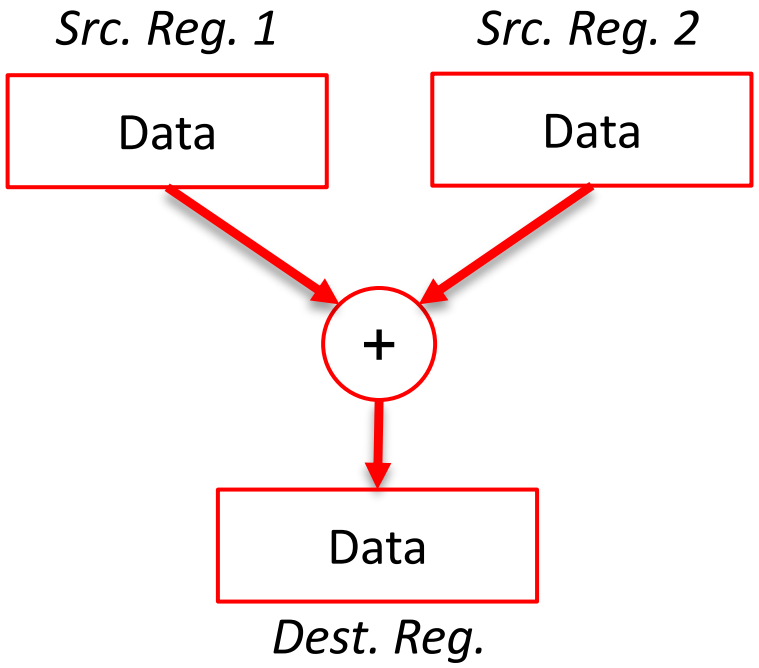


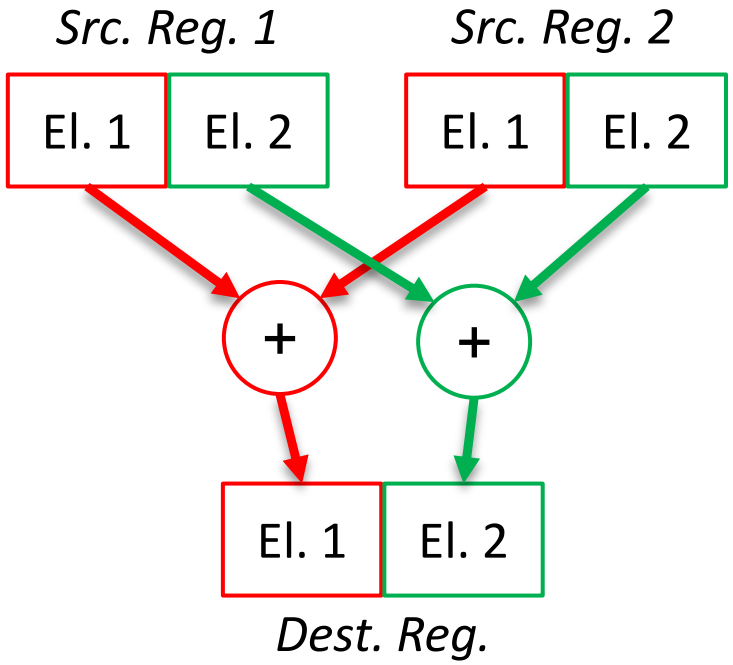
- Data path in CPU is 32 bits wide
  - Registers, arithmetic/logic unit, memory interface
- Interpret and process those 32 bits as multiple elements of a vector
  - E.g. two 16-bit values, four 8-bit values packed into 32 bits
- Now a single instruction can operate on multiple elements
  - Up to 2x or 4x speed-up
- Available on Cortex-M4, M7

SIMD Data Types and Instructions Available

- 32-bit SIMD in ARMv7-M
  - Operations
    - Add, subtract, multiply, exchange, absolute value, accumulate, select
  - Data types and operation variants
    - Size: 8 bit (byte) or 16 bit (halfword)
    - Signed or unsigned (s, u)
    - Saturating (q): result does not overflow/underflow, but instead is clipped
    - Halving (h): result is divided by two, eliminating overflow possibility
  - Not all operations are available for all data types
  - Full information in Armcc User Guide, Ch. 12 (ARMv6 SIMD Instruction Intrinsics)
- Advanced SIMD defined in ARMv7-A, included in ARMv8-A
  - Very-high-performance 128-bit data path
  - Implemented in Neon unit in CPU, pipelined
  - Adds SIMD register file with sixteen 128-bit "quad" registers
  - Data types
    - 8, 16, 32 bits
    - Signed, unsigned, float (single precision)
  - Operations
    - Far too many to list here!

SIMD (Mini-Vector) Concepts



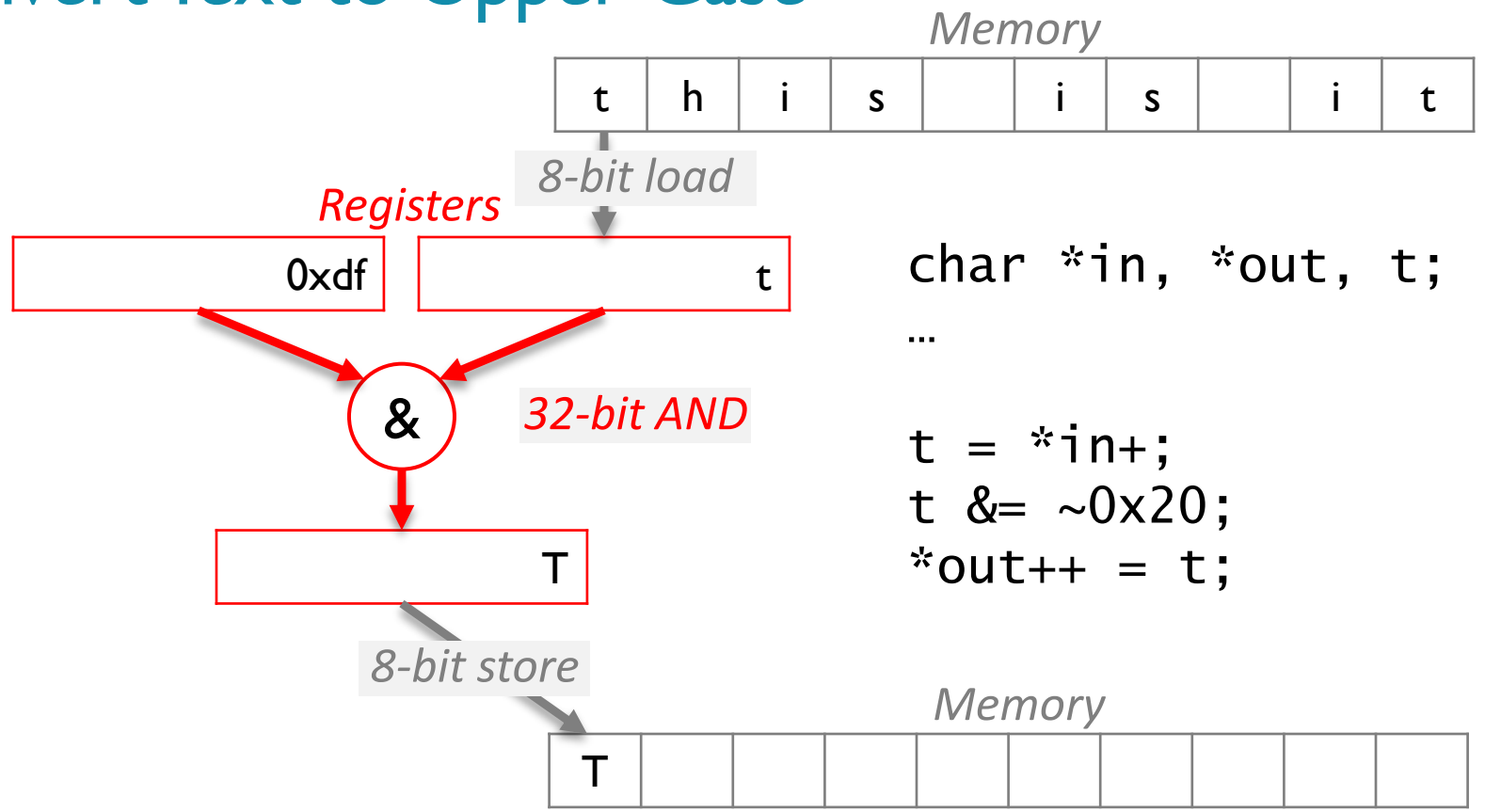


- Data path (registers, ALU, buses, etc.) is 32 bits wide
  - Can we pack multiple data items into a single 32-bit value?

- SIMD: Single Instruction is applied to Multiple Data values simultaneously
  - One register has multiple lanes, each holding a data value
  - 32 1-bit lanes, four 8-bit lanes, two 16-bit lanes?

Example Application: Convert Text to Upper Case

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
@	64	0100	0x40	`	96	0140	0x60
A	65	0101	0x41	a	97	0141	0x61
B	66	0102	0x42	b	98	0142	0x62
C	67	0103	0x43	c	99	0143	0x63
D	68	0104	0x44	d	100	0144	0x64
E	69	0105	0x45	e	101	0145	0x65
F	70	0106	0x46	f	102	0146	0x66
G	71	0107	0x47	g	103	0147	0x67
H	72	0110	0x48	h	104	0150	0x68
I	73	0111	0x49	i	105	0151	0x69
J	74	0112	0x4a	j	106	0152	0x6a
K	75	0113	0x4b	k	107	0153	0x6b
L	76	0114	0x4c	l	108	0154	0x6c
M	77	0115	0x4d	m	109	0155	0x6d

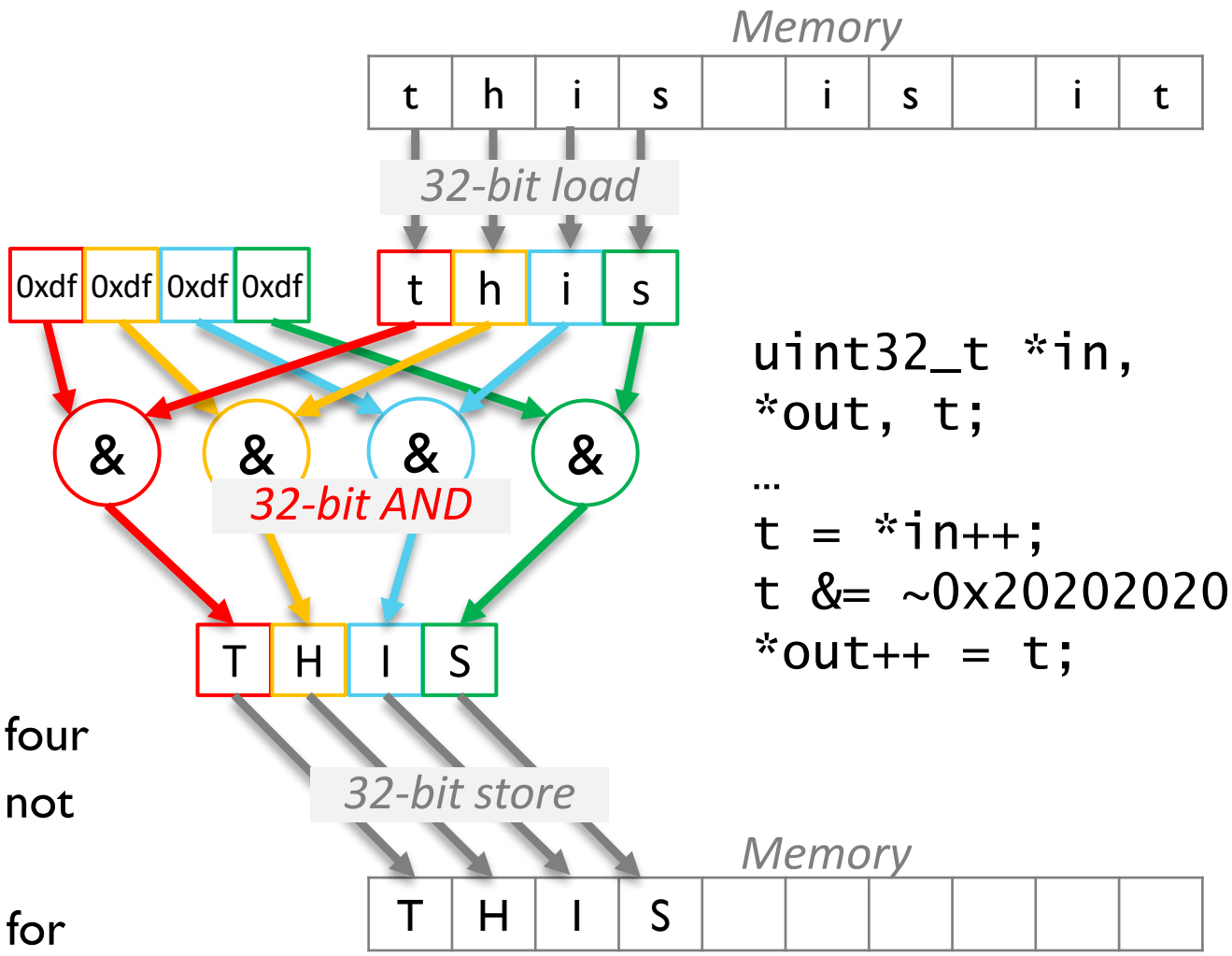



```
char *in, *out, t;
…

t = *in+;
t &= ~0x20;
*out++ = t;
```

- Text represented with 8-bit ASCII data
- Clear bit 6 to convert from lower to upper case
  - AND with ~0x20 (0xdf)
- Converts one character at a time
- Processor has 32-bit data path, which can hold four 8-bit lanes. Can we do better?

Example Application: Convert Text to Upper Case

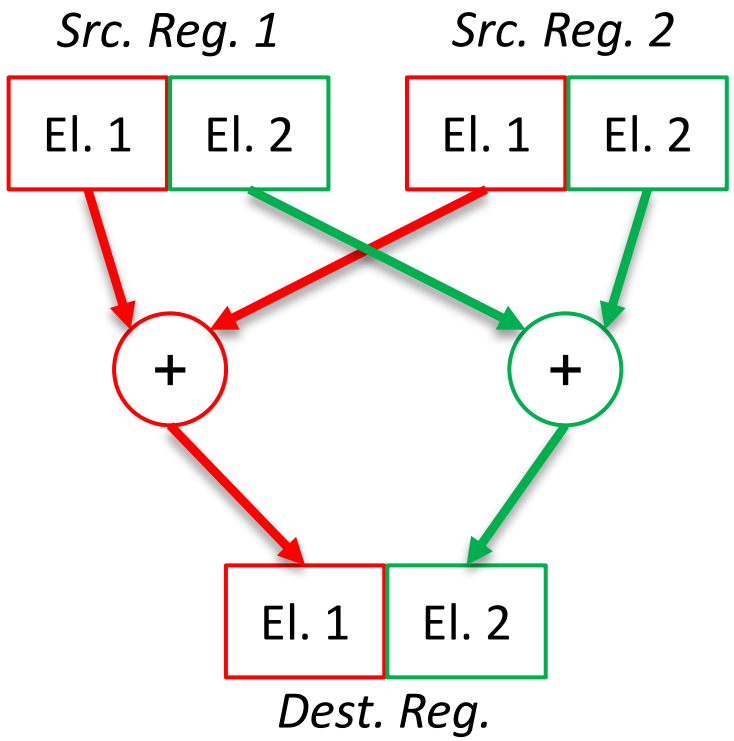
- Have processor interpret data four bytes at a time (as uint32_t)
  - Data in memory is arranged sequentially, no reorganization needed
  - Convert inputs, temps, outputs to 32 bits
  - Replicate constant ~0x20 across all lanes (~0x20202020)
  - Pointers will automatically be incremented by 4 instead of 1
- Restrictions
  - Assumes number of data items is multiple of four
  - Will also convert some symbols if inputs are not tested to be characters: {→[, }→], |→\,~→^
  - Is AND a special operation, or will this work for every operation?



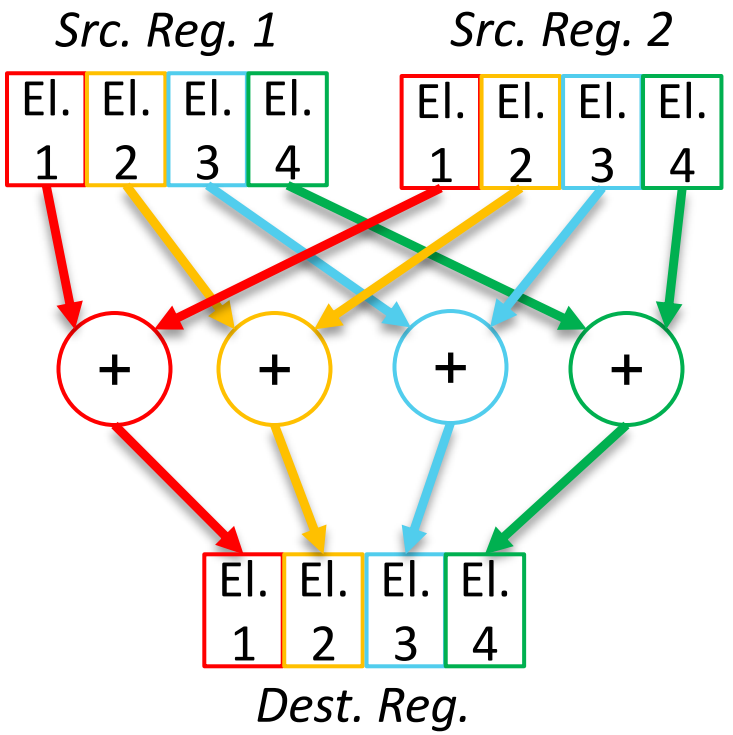

```
uint32_t *in,
*out, t;
…
t = *in++;
t &= ~0x20202020;
*out++ = t;
```

Generalization to Other Operations?

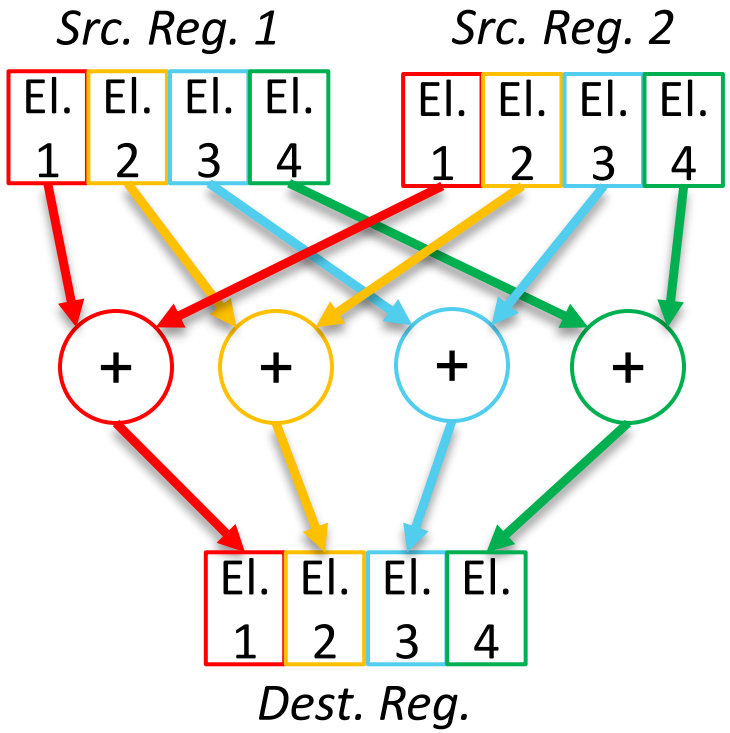
- This approach works if lanes are independent (one lane cannot affect another)
- Independent lanes give inherently SIMD operations
  - AND, NAND, OR, XOR, NOR, NOT
- Other instructions have dependence between lanes, preventing SIMD operations
  - Rotate, shift
  - Add (carry), subtract (borrow), multiply, divide
- Need special versions of these operations
- ARMv7-M provides some 32-bit SIMD instructions based on ADD, SUB, and MUL



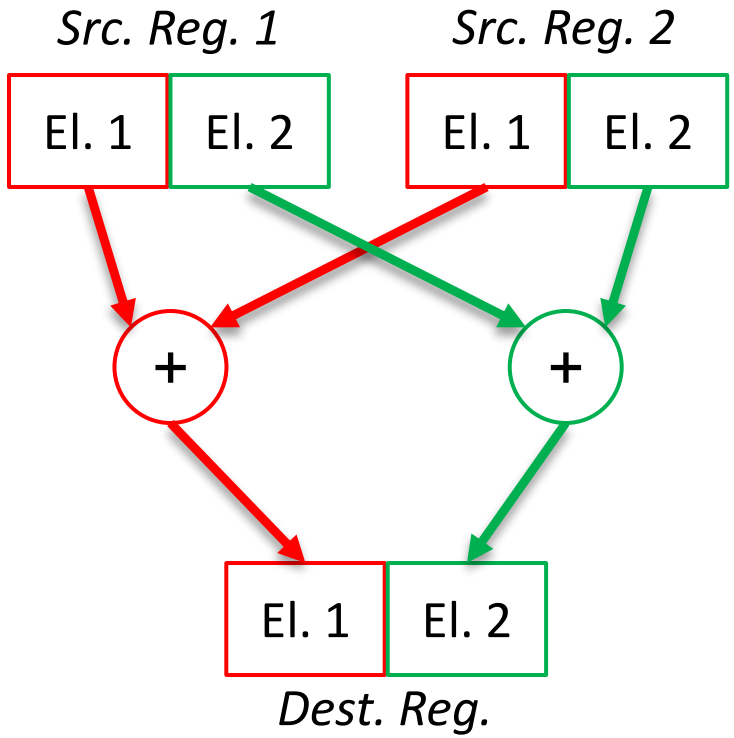
ARMV7-M SIMD AND DSP SUPPORT



Data Sizes for 32-bit SIMD Instructions



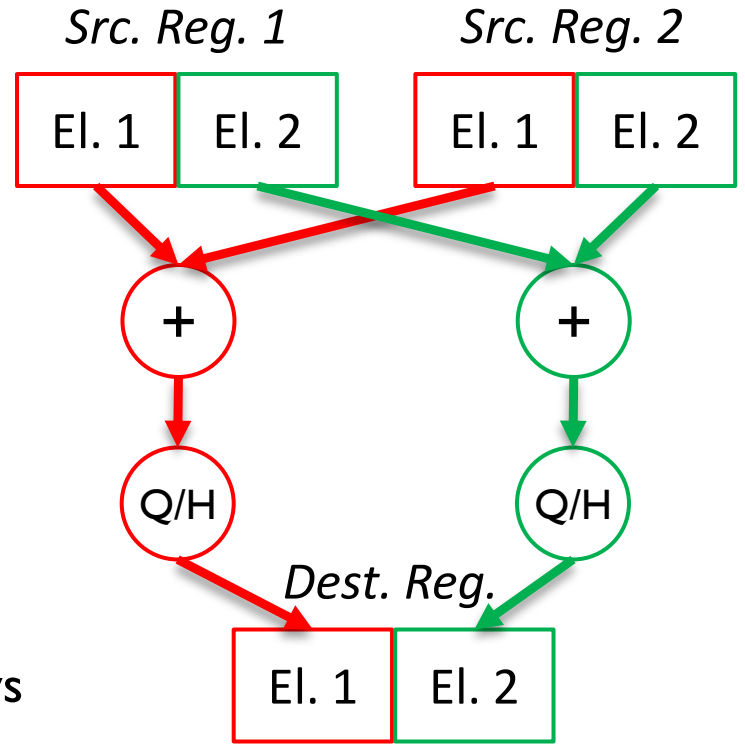
- Four eight-bit lanes



- Two sixteen-bit lanes

32-bit SIMD Arithmetic Instructions

- Basic Instructions
  - ADD[8|16]: Byte-wise or halfword-wise addition
  - SUB[8|16]: Byte-wise or halfword-wise subtraction
- Result status bits in program status register
  - Four bits GE[0-3], corresponding to each lane
  - SADD, SSUB: sets lane bit to 1 if lane result ≥ 0
  - UADD, USUB: sets lane bit to 1 if lane result overflows or underflows
- Prefixes
  - Signed (S): signed math, updates GE bits
  - Unsigned (U): unsigned math, updates CPSR GE bits
  - Saturating (Q): Limit value to closest valid value
  - Halving (H): Divide result by two

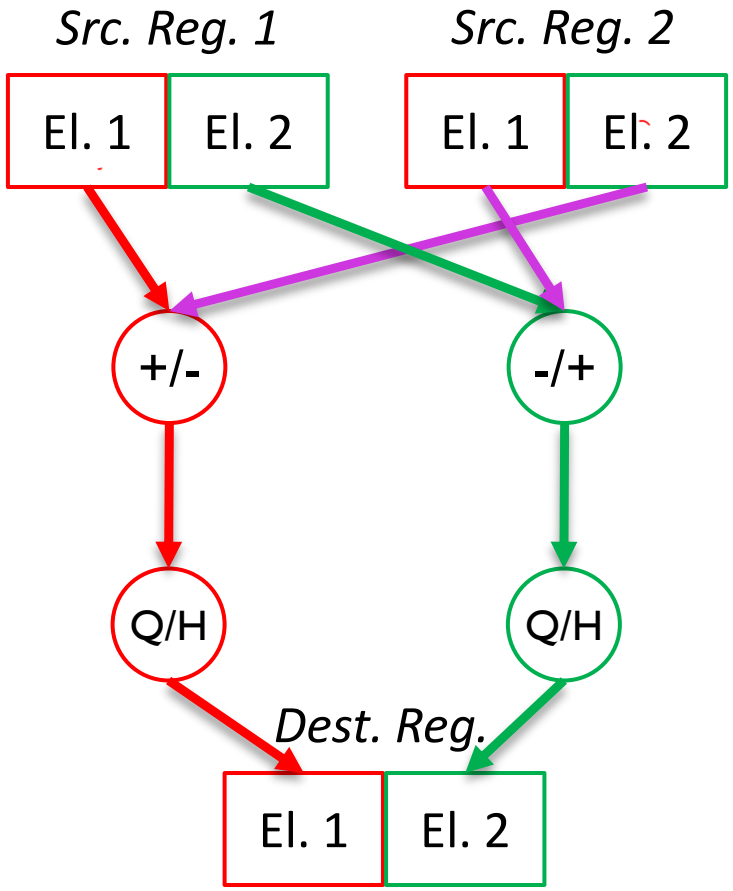


Prefixes for Parallel Instructions	
S	Signed arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
Q	Signed saturating arithmetic
SH	Signed arithmetic, halving results
U	Unsigned arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
UQ	Unsigned saturating arithmetic
UH	Unsigned arithmetic, halving results

32-bit SIMD Arithmetic Instructions

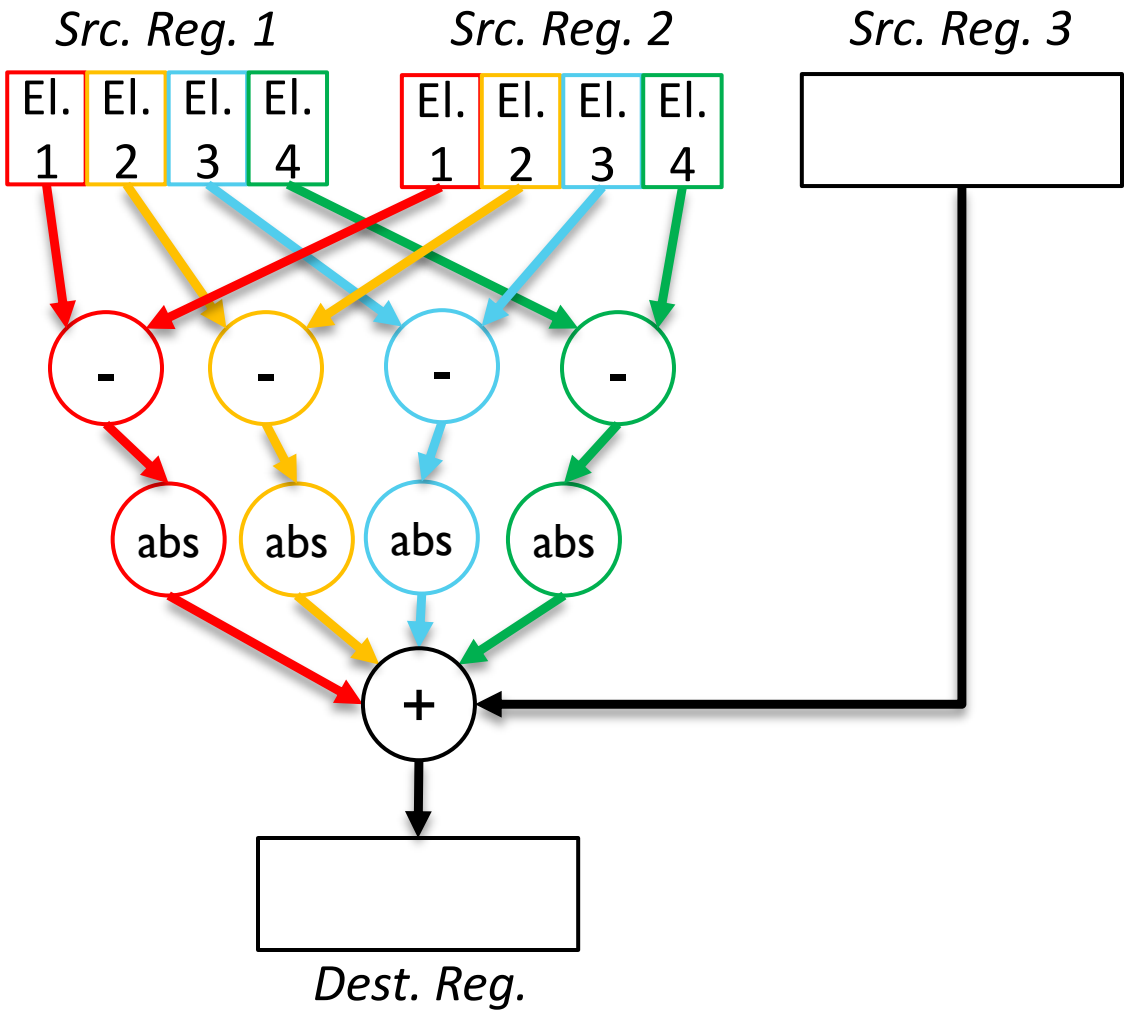
- More Instructions
  - ASX: Halfword-wise exchange, add, subtract
  - SAX: Halfword-wise exchange, subtract, add
- Prefixes
  - Saturating
  - Halving

Prefixes for Parallel Instructions	
S	Signed arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
Q	Signed saturating arithmetic
SH	Signed arithmetic, halving results
U	Unsigned arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
UQ	Unsigned saturating arithmetic
UH	Unsigned arithmetic, halving results



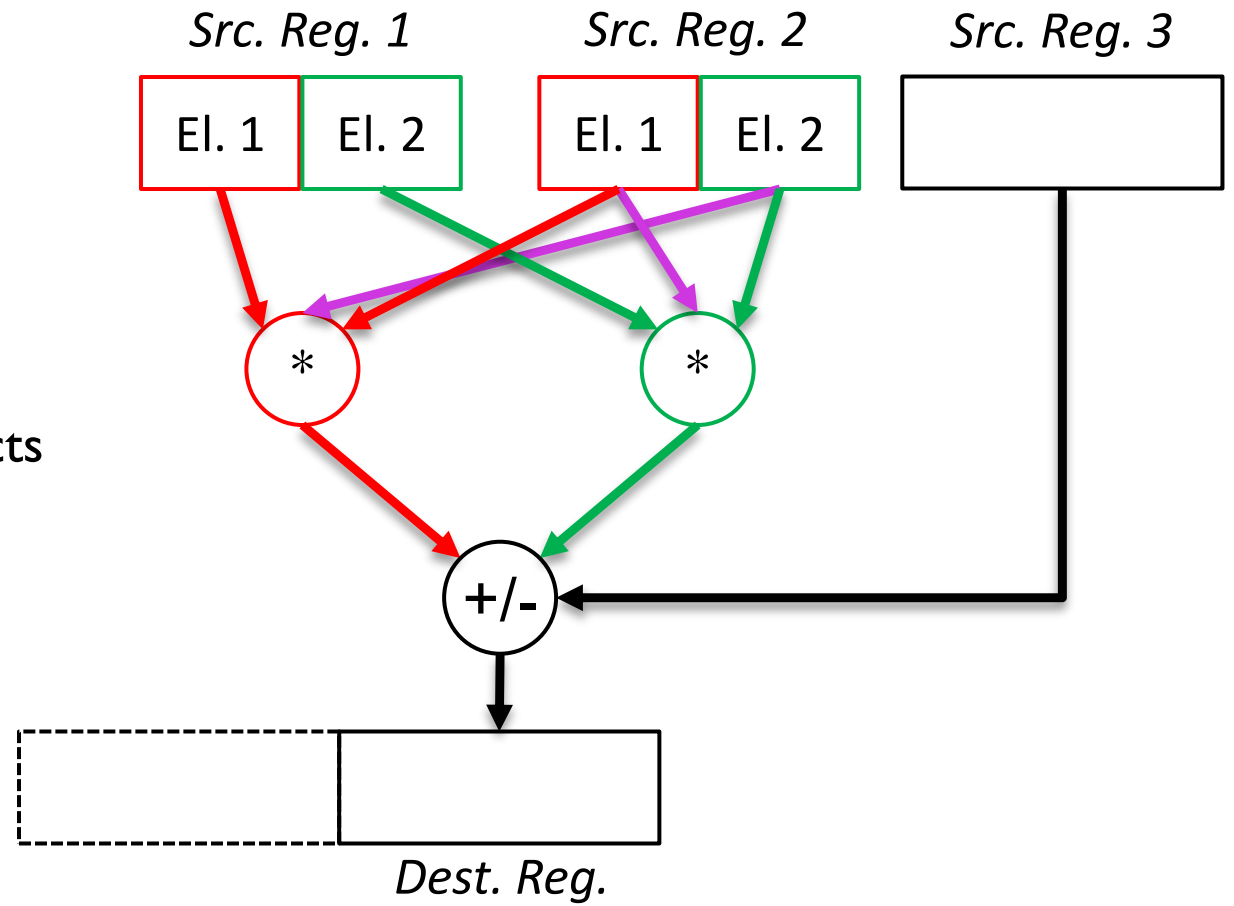
32-bit SIMD Arithmetic Instructions

- More Instructions
  - USAD8: Unsigned sum of absolute differences
  - USADA8: Unsigned sum of absolute differences and accumulate



32-bit SIMD Multiplication Instructions

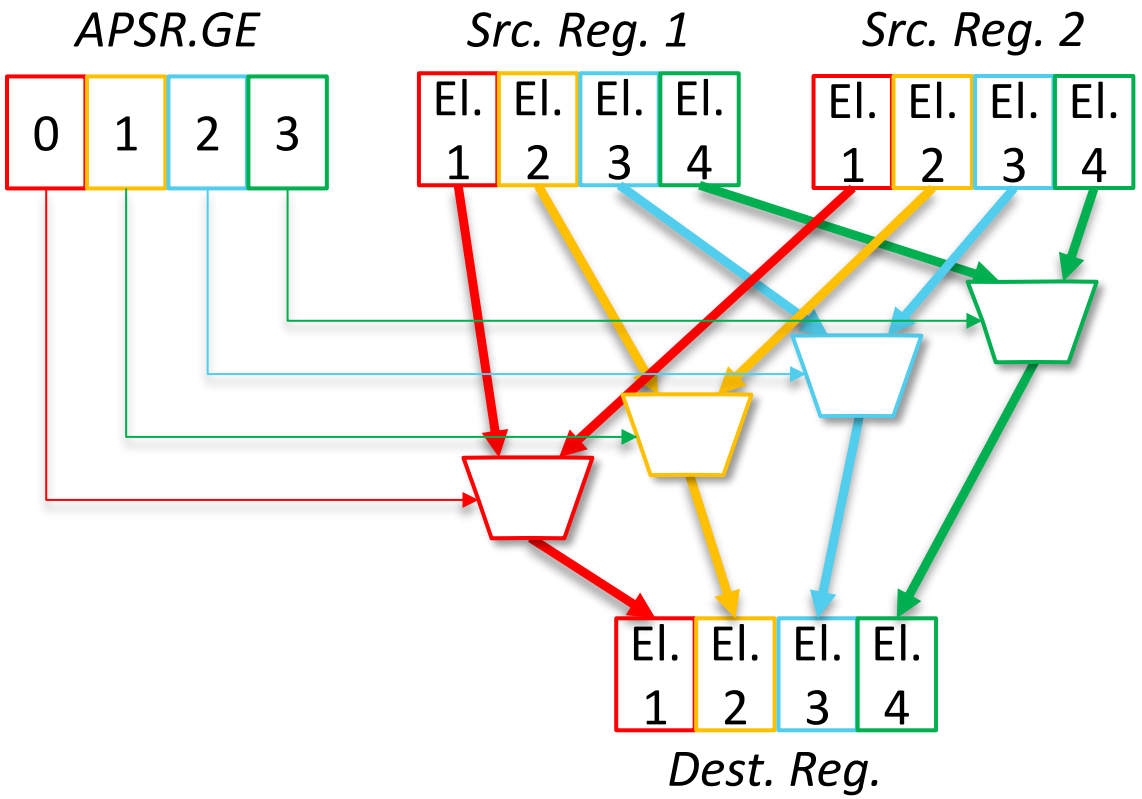
- SM[U|L][A|S]{L}D{X}: Dual halfword signed multiply, add/subtract products
- [U|L] Options
  - U: SMU – Base operation: multiply halfwords, add/subtract products
  - L: SML – Accumulate sum (or difference) of products in 32-bit register
- [A|S] Options
  - A: Add products
  - S: Subtract products
- {L} Option
  - L: Accumulate to 64-bit register
- {X} Option
  - X: Exchange halfwords of one operand before multiplications



32-bit SIMD Miscellaneous Instructions

- Saturation
  - SSAT16: Saturate halfwords to range -2^{n-1} to $2^{n-1}-1$, with n as argument
  - USAT16: Saturate halfwords to range 0 to $2^{n}-1$, with n as argument
- Extraction with extension (and optional addition)
  - UXT{A}B16: extract low byte of each half-word, zero extend to 16 bits, optional add to first operand
  - SXT{A}B16: extract low byte of each half-word, sign extend to 16 bits, optional add to first operand
- Packing
  - PKHBT: pack halfword, bottom and left-shifted top (LSL)
  - PKHTB: pack halfword, top and right-shifted bottom (ASR)

32-bit SIMD Miscellaneous Instructions: Selection



- SEL: Select bytes based on GE (greater than or equal) flags
  - APSR GE flags updated by [U|S][ADD|SUB][8|16]

- Example 1:
  - SADD16 R0, R1, R2: Signed halfword add
  - SEL R3, R4, R5: Select
    - R3[15:0] = R4[15:0] if low-word result (of SADD16) is ≥ 0, else R5[15:0]
    - R3[31:16] = R4[31:16] if high-word result (of SADD16) is ≥ 0, else R5[31:16]
- Example 2:
  - UADD8 R0, R1, R2: Signed byte add
  - SEL R3, R4, R5
    - R3[7:0] = R4[7:0] if UADD8 low byte result overflowed, else R5[7:0]
    - Similar for other bytes

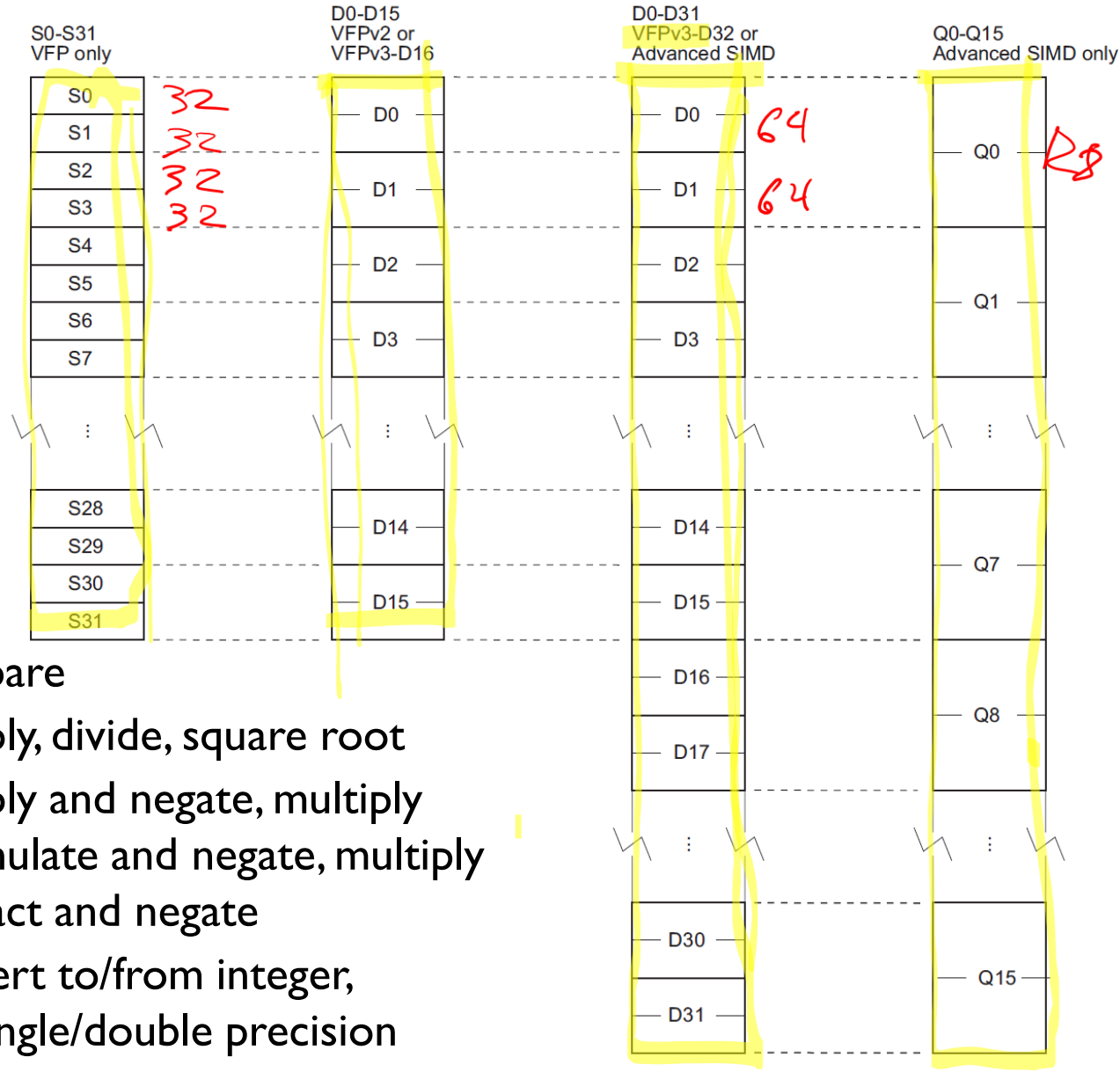
References

- MDK Armcc User Guide: DUI0375
  - Chapter 12: ARMv6 SIMD Instruction Intrinsics
- ARM C Language Extensions (ACLE): IHI0053 (different syntax, not used for armcc v5)
  - 9.3: 16-bit multiplications
  - 9.4: Saturating intrinsics
  - 9.5: 32-bit SIMD intrinsics
  - 11: Instruction generation

FLOATING-POINT MATH

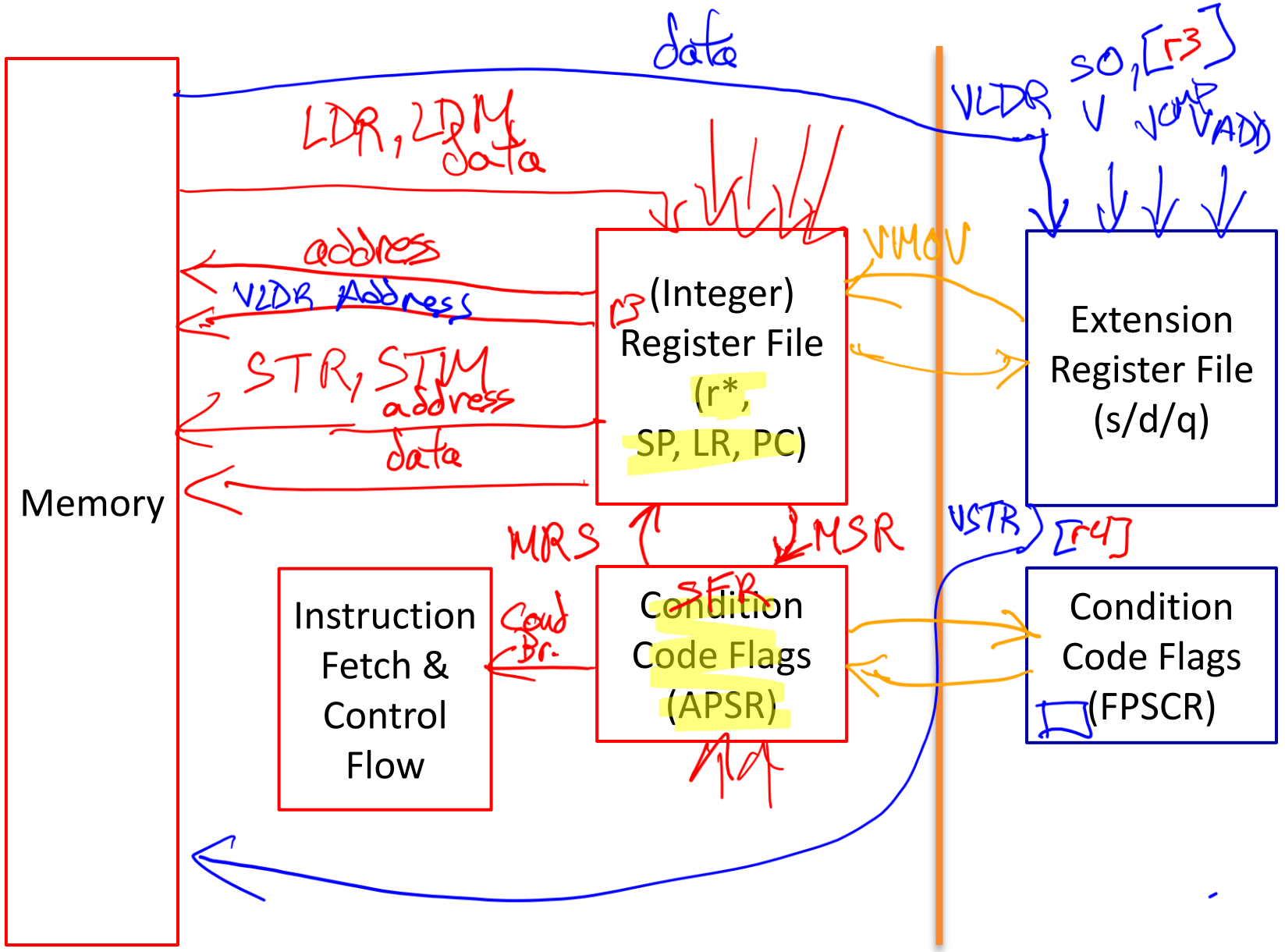
Support for Floating-Point Math

- Extension register file added
  - Support for floating point data
    - S0-S31: Single-precision (32 bits)
    - D0-D15/D31: Double-precision (64 bits)
  - Also used for Advanced SIMD
- Other registers added
  - FPSCR: status and control register. Condition code flags N, Z, C, V.
  - Others too: ID, exception, features
- Floating-point instructions (V…)
  - Move immediate value into register
  - Copy register
  - Load/store one or multiple 32-bit registers from/to memory
  - Add, subtract, negate, absolute value
  - Compare
  - Multiply, divide, square root
  - Multiply and negate, multiply accumulate and negate, multiply subtract and negate
  - Convert to/from integer, half/single/double precision



Partition Between Integer and VFP/Adv. SIMD

- Register files and instructions are partitioned
  - Can't access any register from any instruction
  - Need special instructions to cross the boundary
- Need to use integer registers as pointers to access memory



Load/Store

Vector Load Multiple	*VLDM* on page A8-626	Load 1-16 consecutive 64-bit registers (Adv. SIMD and VFP) Load 1-16 consecutive 32-bit registers (VFP only)
Vector Load Register	*VLDR* on page A8-628	Load one 64-bit register (Adv. SIMD and VFP) Load one 32-bit register (VFP only)
Vector Store Multiple	*VSTM* on page A8-784	Store 1-16 consecutive 64-bit registers (Adv. SIMD and VFP) Store 1-16 consecutive 32-bit registers (VFP only)
Vector Store Register	*VSTR* on page A8-786	Store one 64-bit register (Adv. SIMD and VFP) Store one 32-bit register (VFP only)

Register Transfer Instructions

Copy byte, halfword, or word from ARM core register to extension register	*VMOV (ARM core register to scalar)* on page A8-644
Copy byte, halfword, or word from extension register to ARM core register	*VMOV (scalar to ARM core register)* on page A8-646
Copy from single-precision VFP register to ARM core register, or from ARM core register to single-precision VFP register	*VMOV (between ARM core register and single-precision register)* on page A8-648
Copy two words from ARM core registers to consecutive single-precision VFP registers, or from consecutive single-precision VFP registers to ARM core registers	*VMOV (between two ARM core registers and two single-precision registers)* on page A8-650
Copy two words from ARM core registers to doubleword extension register, or from doubleword extension register to ARM core registers	*VMOV (between two ARM core registers and a doubleword extension register)* on page A8-652
Copy from Advanced SIMD and VFP extension System Register to ARM core register	*VMRS* on page A8-658 *VMRS* on page B6-27 (system level view)
Copy from ARM core register to Advanced SIMD and VFP extension System Register	*VMSR* on page A8-660 *VMSR* on page B6-29 (system level view)

Data Processing I

Absolute value	*VABS* on page A8-532
Add	*VADD (floating-point)* on page A8-538
Compare (optionally with exceptions enabled)	*VCMP, VCMPE* on page A8-572
Convert between floating-point and integer	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
Convert between floating-point and fixed-point	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
Convert between double-precision and single-precision	*VCVT (between double-precision and single-precision)* on page A8-584
Convert between half-precision and single-precision	*VCVTB, VCVTT (between half-precision and single-precision, VFP)* on page A8-588

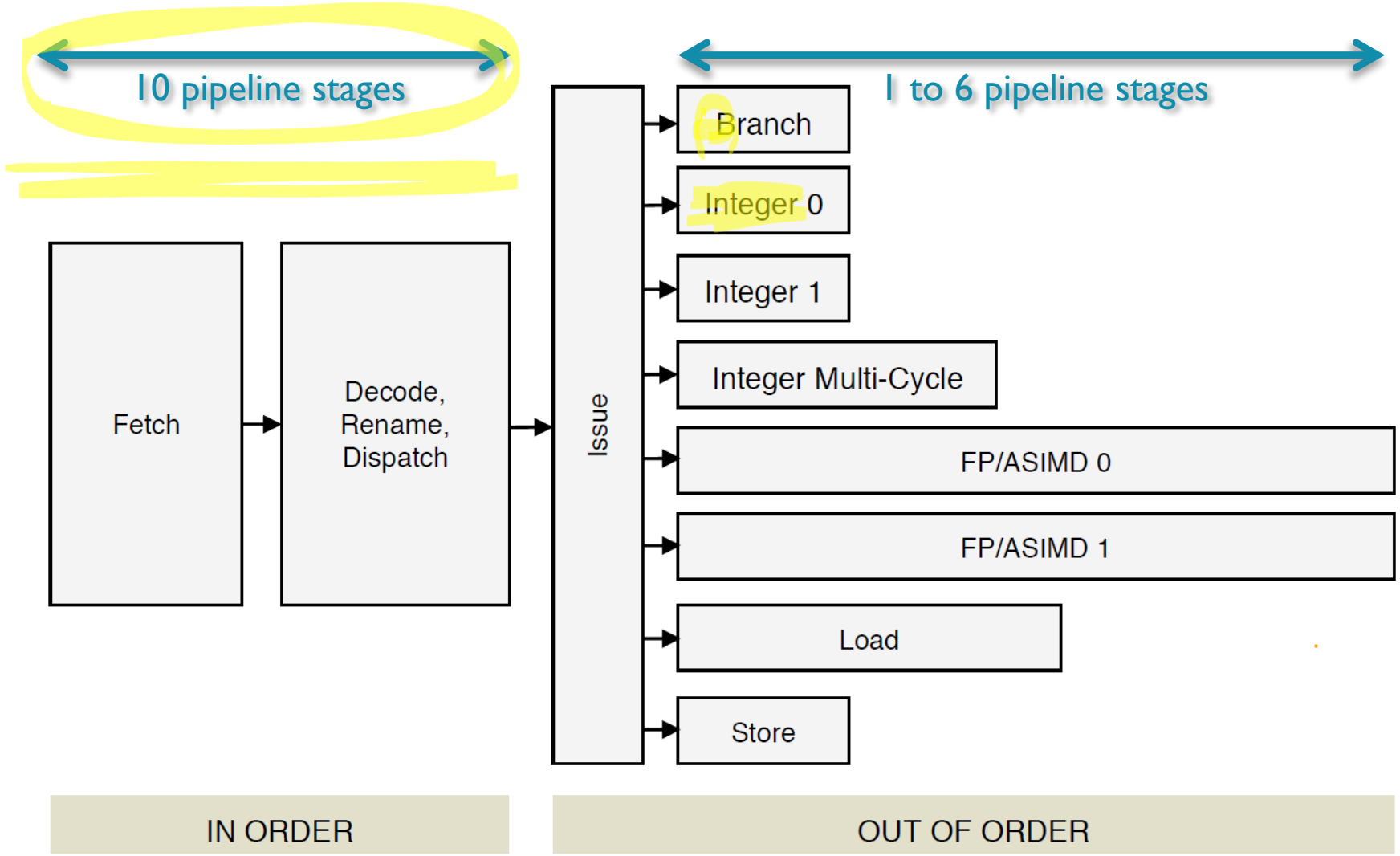
Data Processing 2

Divide	*VDIV* on page A8-590
Multiply Accumulate, Multiply Subtract	*VMLA, VMLS (floating-point)* on page A8-636
Move immediate value to extension register	*VMOV (immediate)* on page A8-640
Copy from one extension register to another	*VMOV (register)* on page A8-642
Multiply	*VMUL (floating-point)* on page A8-664
Negate (invert the sign bit)	*VNEG* on page A8-672
Multiply Accumulate and Negate, Multiply Subtract and Negate, Multiply and Negate	*VNMLA, VNMLS, VNMUL* on page A8-674
Square Root	*VSQRT* on page A8-762
Subtract	*VSUB (floating-point)* on page A8-790

Nice instructions.

But how long do they take to execute?

Cortex-A72 Instruction Processing Pipeline



New FP/ASIMD units are only 6 pipe stages long (vs. 10 on A57)

Instruction Latencies

Integer Instruction Type	Result Latency (min. to max. cycles, assuming L1 cache hits)
Branch	1
Arithmetic, Logic	1-2
Move, Shift	1-2
Divide	4-20. Blocking
Multiply	3-6
Saturating and Parallel Arithmetic	2-4 Int SIMD
Misc. Data Processing	1-4
Load	4-5
Store	1-3

3 to 12 pipeline stages

Other Instruction Type	Result Latency (min. to max. cycles, assuming L1 cache hits)
FP Data Processing	3-4
FP Divide	6-11 (SP), 6-18 (DP). Blocking
FP Square Root	6-17 (SP), 6-32 (DP). Blocking
FP Load	5-6
FP Store	1-4
ASIMD Integer	3-5
ASIMD FP	3-7
ASIMD Misc.	3-8
ASIMD Load	5-9
ASIMD Store	1-4
Crypto	3-6
CRC	2