TURIN POLYTECHNIC
UNIVERSITY IN
TASHKENT

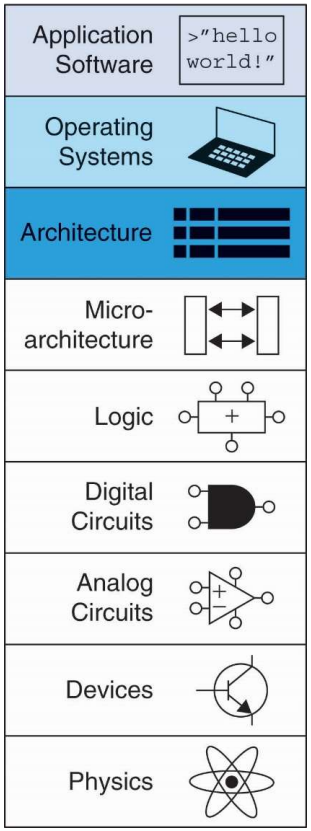
Fundamentals of Computer Architecture

ARM Architecture and Assembly

Teacher: Lobar Asretdinova

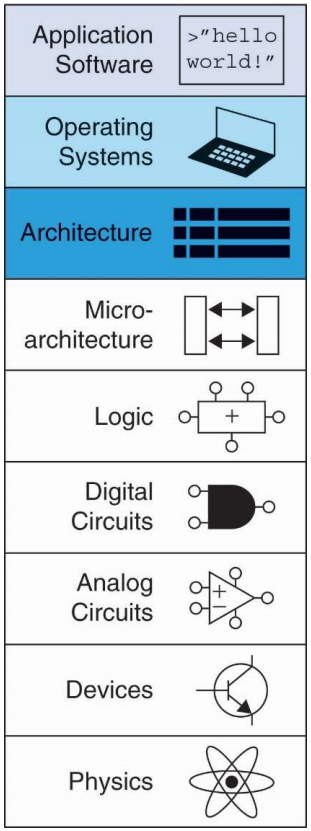
:: Topics

- **Introduction**
- **Assembly Language**
- **Machine Language**
- **Programming**
- **Addressing Modes**



Introduction

- **Jumping up a few levels of abstraction**
  - **Architecture:** programmer's view of computer
    - Defined by **instructions** & **operand locations**
  - **Microarchitecture:** how to implement an architecture in hardware



Instructions

- **Commands in a computer's language**
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines, etc.

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines,, etc.

Once you've learned one architecture, it's easier to learn others

A Simple Processor

We will only need a few simple components:

- **Memories** – to store our program (instructions) and data
- **A register file** – instructions will read their operands from the register file and also write their results to it.
- **Registers**, an **ALU** and **adders**
- **Decode and control logic**

A Simple (32-bit) Processor

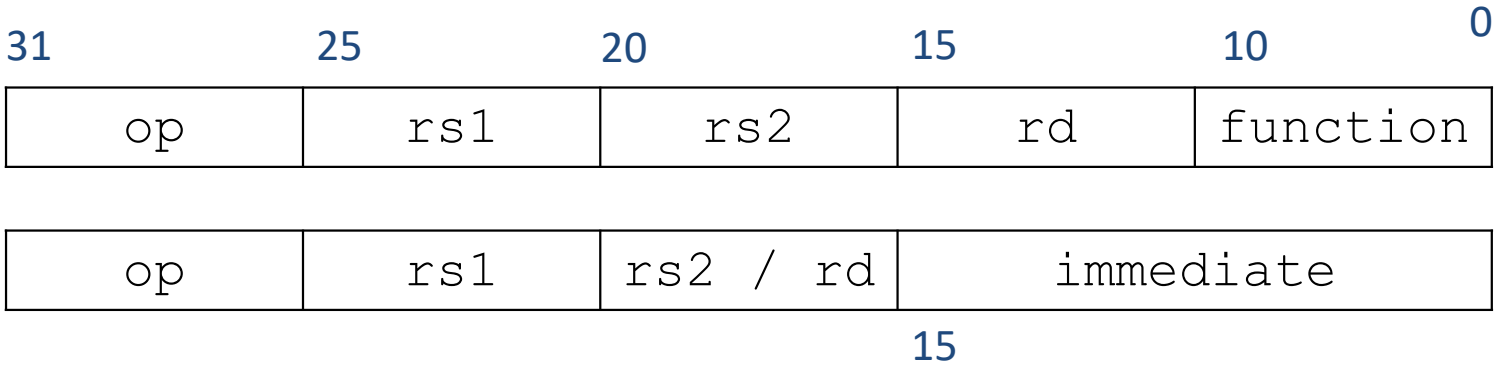
- Let's assume all our instructions are encoded in 32-bits/
- Our registers and datapath are also 32-bits wide.
- Memory is accessed with a 32-bit address and returns 32-bit data.
- Our processor has 32 registers, hence we must use 5-bits to identify a particular register (as 2^5 =32).

A Processor Datapath – Encoding Instructions

- A simple data processing instruction may have the following format, where `Operand2` may be a register or immediate value.

```
Instruction Rd, Rs, Operand2
```

- Given 32-bits to encode our instructions, we may invent two simple instruction encoding formats for our processor, e.g.:

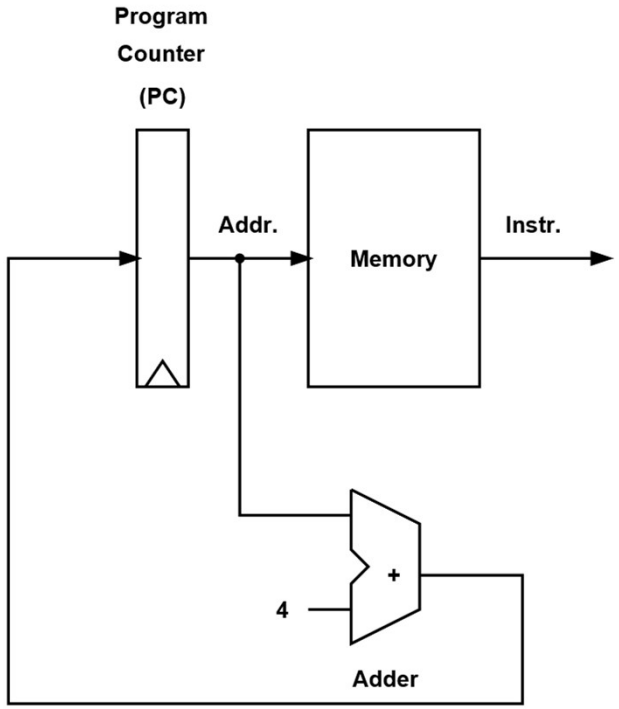


A Processor Datapath – PC and Instruction Memory

Our Program Counter (PC) stores the address of the instruction currently being fetched from memory.

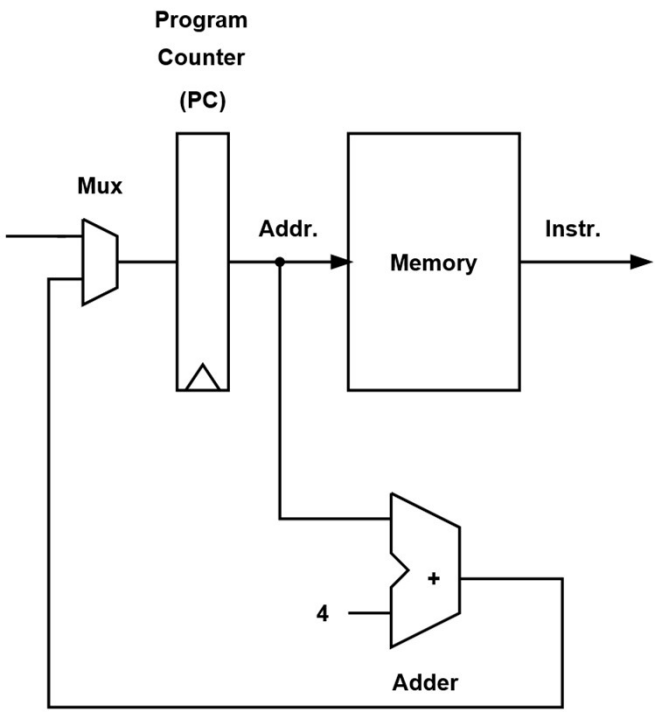
This simple datapath increases the PC by 4 (bytes) on each cycle and only allows us to read each 32-bit instruction in turn.

We need to add more logic to actually compute, handle branches, provide registers, access data memory, etc.



A Processor Datapath – Add mux to Allow Jump/Branch

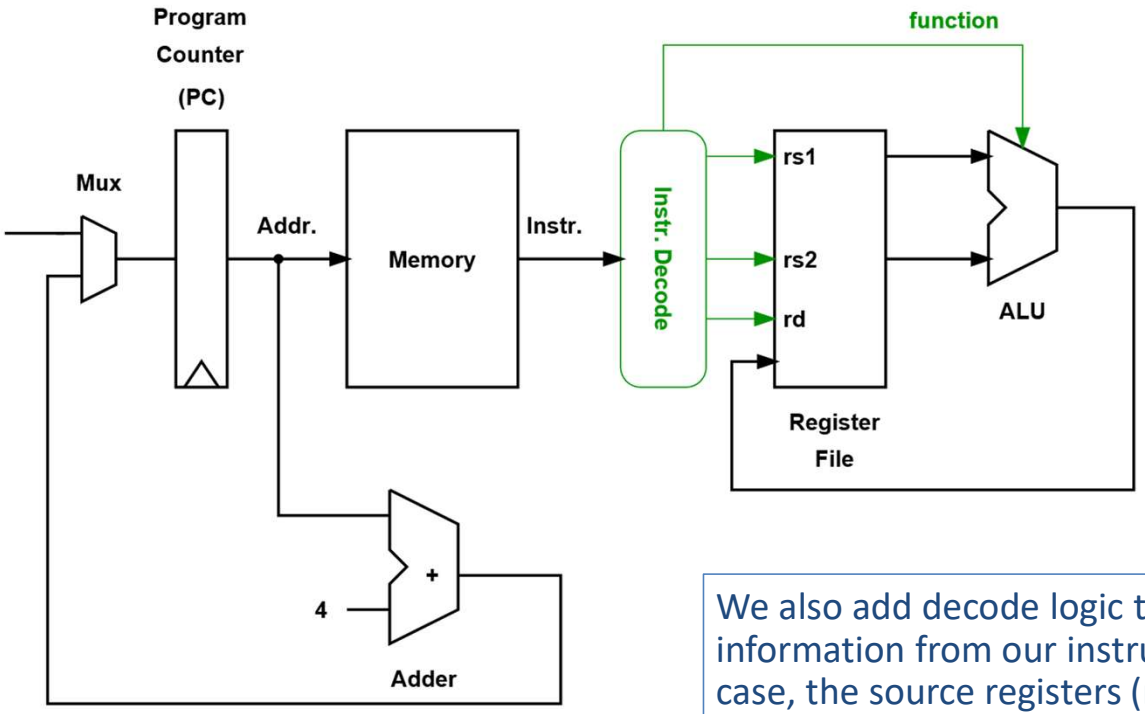
We add a multiplexor (mux) to allow us to provide a branch target address, i.e., the PC value following a taken branch.



A Processor Datapath – Add a Register File and ALU

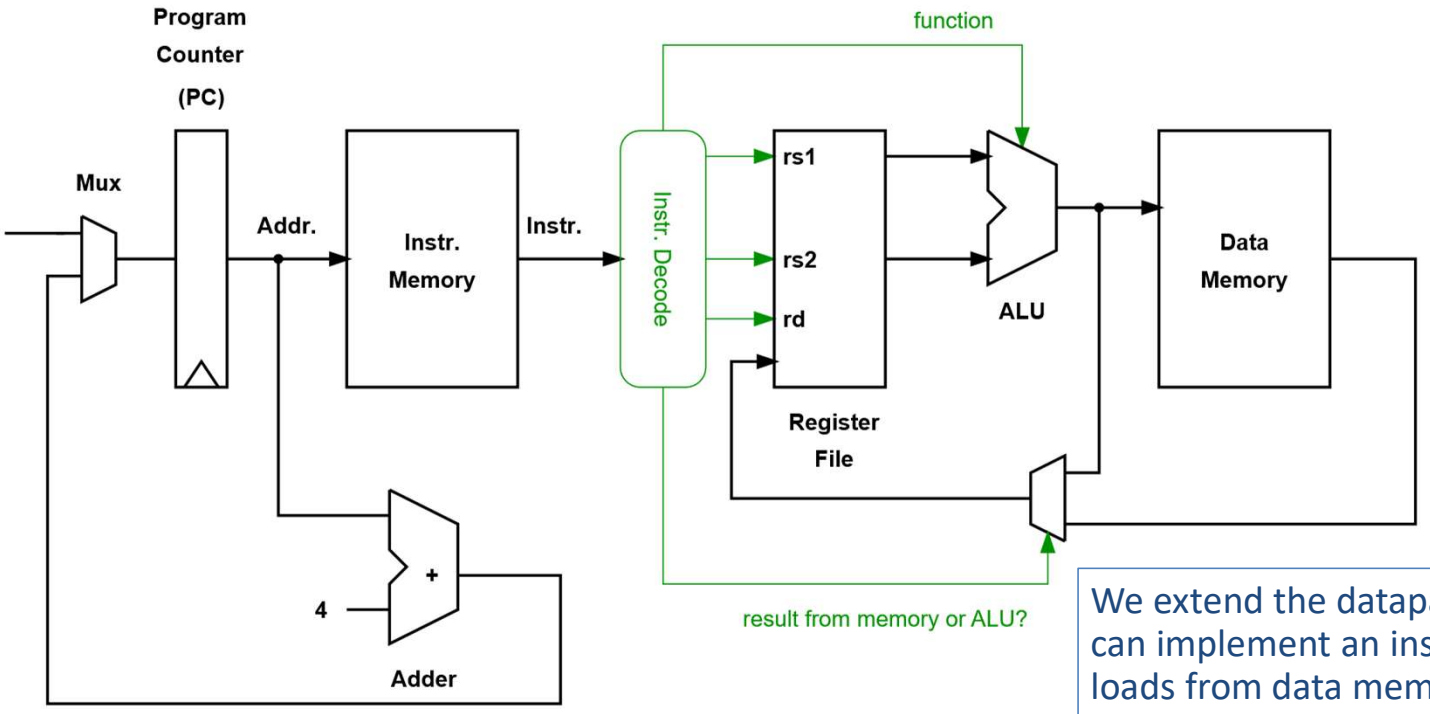
We could now execute a simple sequence of ALU operations.

Although, we do not yet have access to data memory and can't branch!



We also add decode logic that extracts information from our instruction; in this case, the source registers (rs1, rs2), destination register (rd), and the ALU function to be performed.

A Processor Datapath – Loading from Data Memory

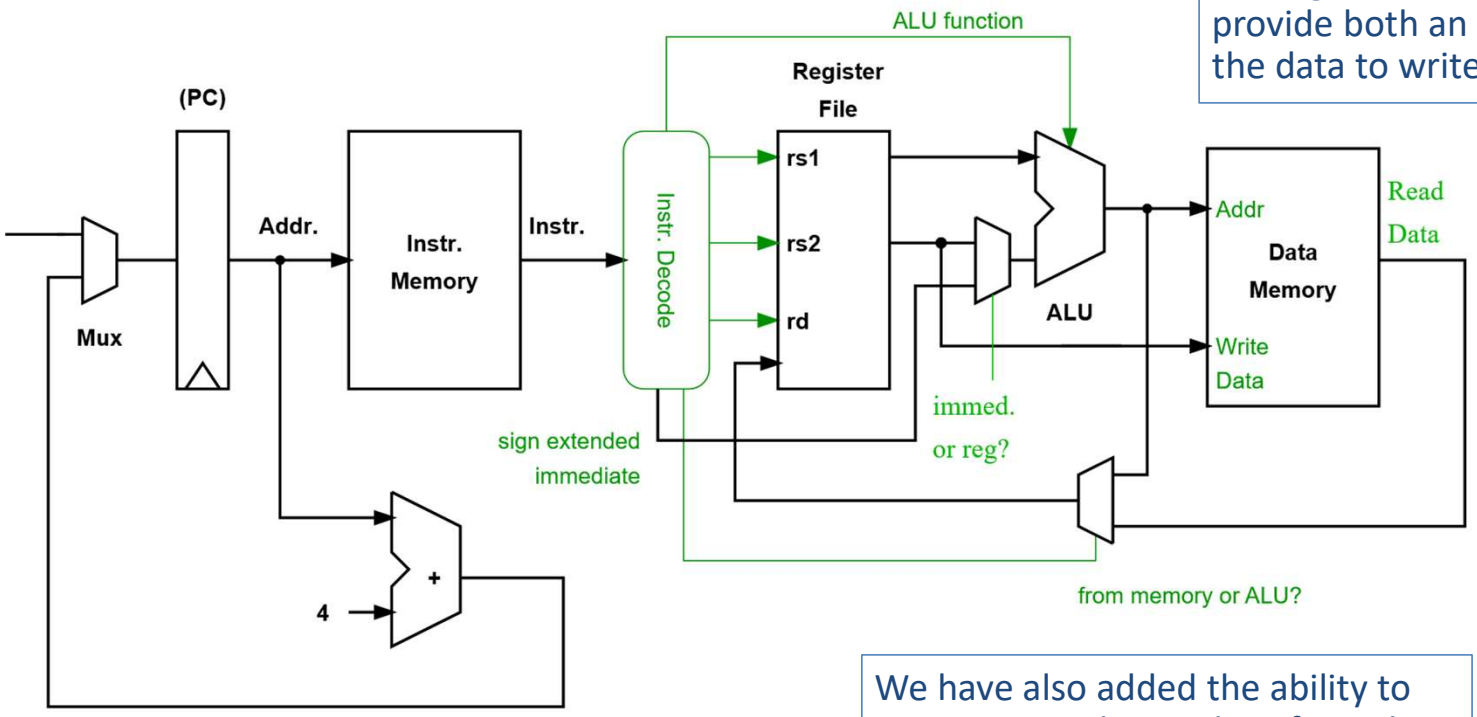


We extend the datapath so we can implement an instruction that loads from data memory.

The effective address could be the sum of the two source registers.

A Processor Datapath – Storing to Data Memory

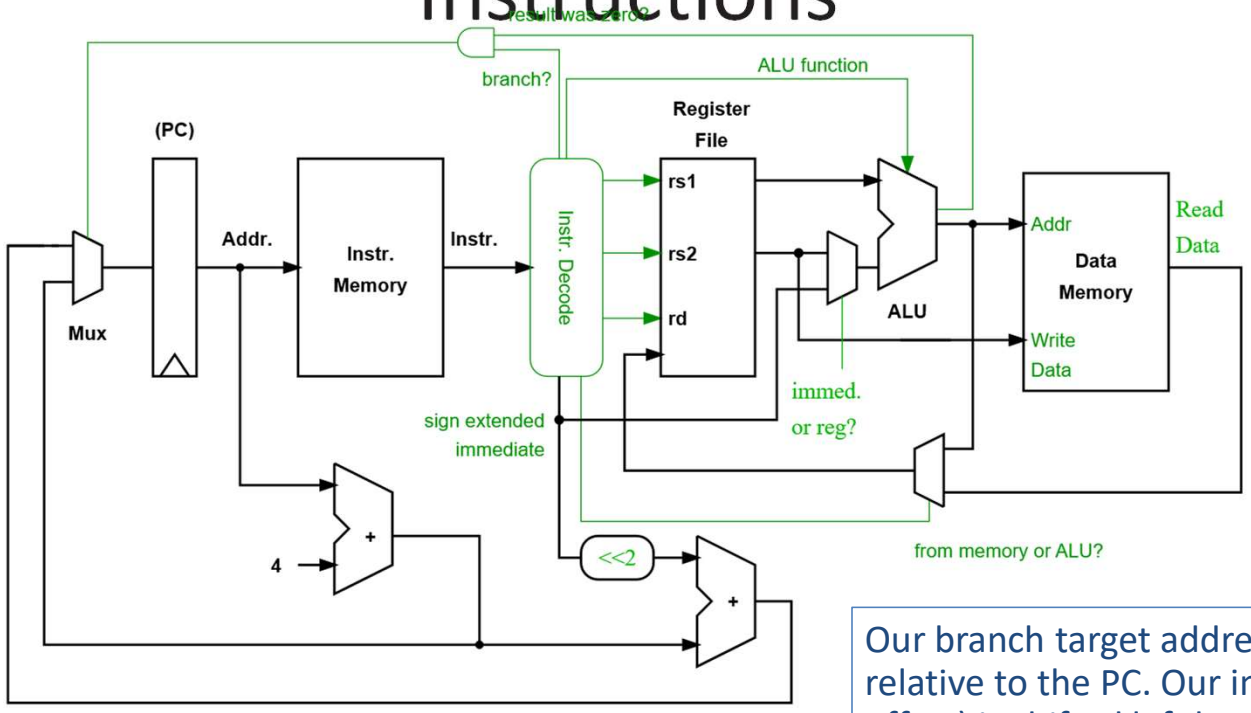
Storing to memory requires we provide both an address and the data to write.



We have also added the ability to extract immediate values from the instructions and use them in ALU operations or address calculations.

A Processor Datapath – Supporting Branch Instructions

Here, we assume a simple "branch if equal to zero" instruction.



Our branch target address is computed relative to the PC. Our immediate (the offset) is shifted left by two (i.e., multiplied by 4) as all instructions are 32-bits.

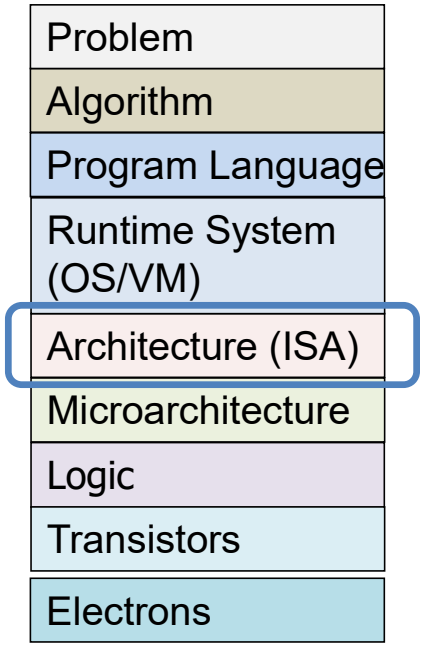
Instruction Set Architecture (ISA)

- ISA is typically seen as the contract between software and hardware.

Instruction Set Architecture (ISA)

- Instruction Set Architecture
  - instructions
  - registers
  - memory addressing
  - addressing modes
  - etc.

The computing abstraction stack



The Instruction Set

The instruction set defines what information can be passed from the compiler to the hardware - what hardware details are exposed to software and what is hidden.

This raises a number of questions:

- At what level do we draw this HW/SW dividing line or interface?
- What other information might it be useful to pass to the hardware to help simplify it?
- How do we ensure good code density?

The RISC Approach

- The RISC approach aims to ensure that we **make the common-case fast** by carefully selecting the most useful instructions and addressing modes, etc.
- Instructions are designed to make good use of the register file.
- A RISC ISA is designed to ensure a simple high-performance implementation is possible.

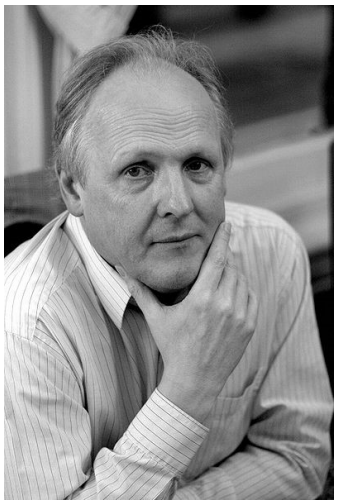
Instruction Set Architecture

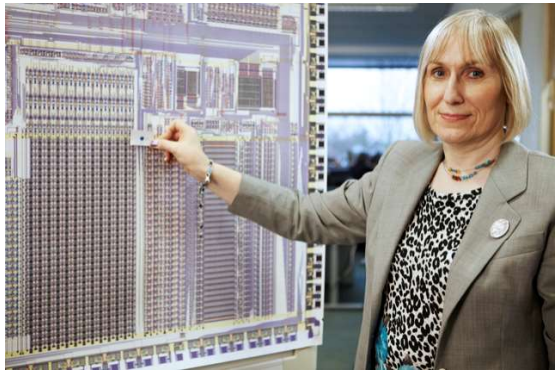
Common features of RISC instruction sets:

- Fixed length instruction encodings (or a small number of easily decoded formats)
- Each instruction follows similar steps when being executed.
- Access to data memory is restricted to special load/store instructions (a so-called **load/store architecture**).

Arm1: The First Arm Processor (1985)

- Arm: **A**dvanced **R**ISC **M**achine (Arm)
- The first Arm processor was designed by Sophie Wilson and Prof. Steve Furber. It was inspired by early research papers from Berkeley and Stanford on RISC.
- Arm1
  - 25,000 transistors
  - 3-stage pipeline
  - 8 MHz clock
  - No on-chip cache





Prof. Steve Furber (left)[1] and Sophie Wilson (right)[2]

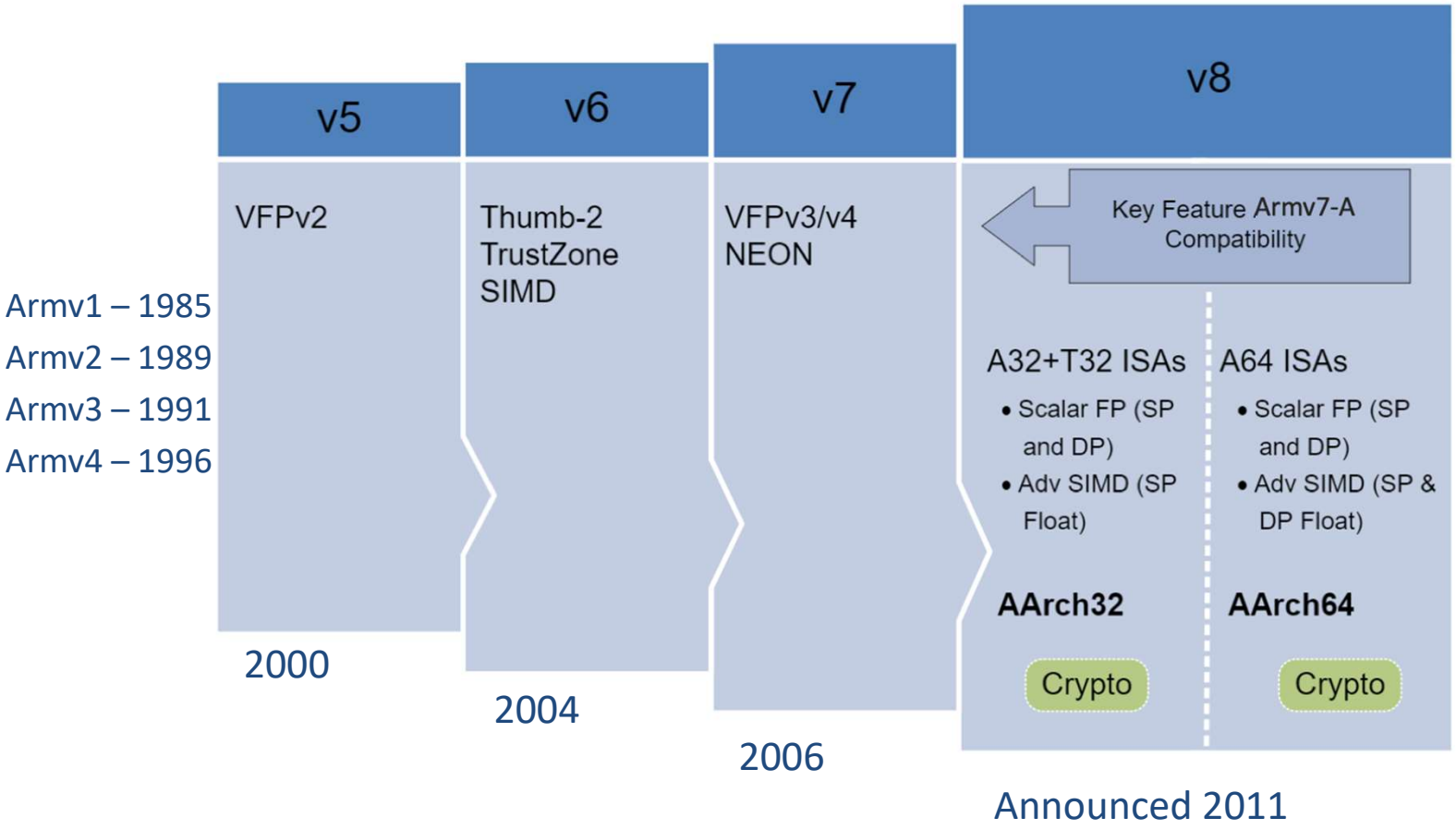
1. By Peter Howkins, CC BY-SA 3.0
2. By Chris Monk, CC BY-SA-2.0

CISC

- Not all processors use RISC ISAs; some are Complex Instruction Set Computers (CISC).
- CISC designs have evolved since the 1970s.
- How are modern CISC machines (e.g., x86) implemented?
  - They first convert the CISC instructions to (possibly numerous) RISC-like micro-ops!
  - Their microarchitectures are otherwise very similar to other modern high-performance processors.

The Armv8-A ISA

Case Study: The Armv8 Architecture



A64 Instructions

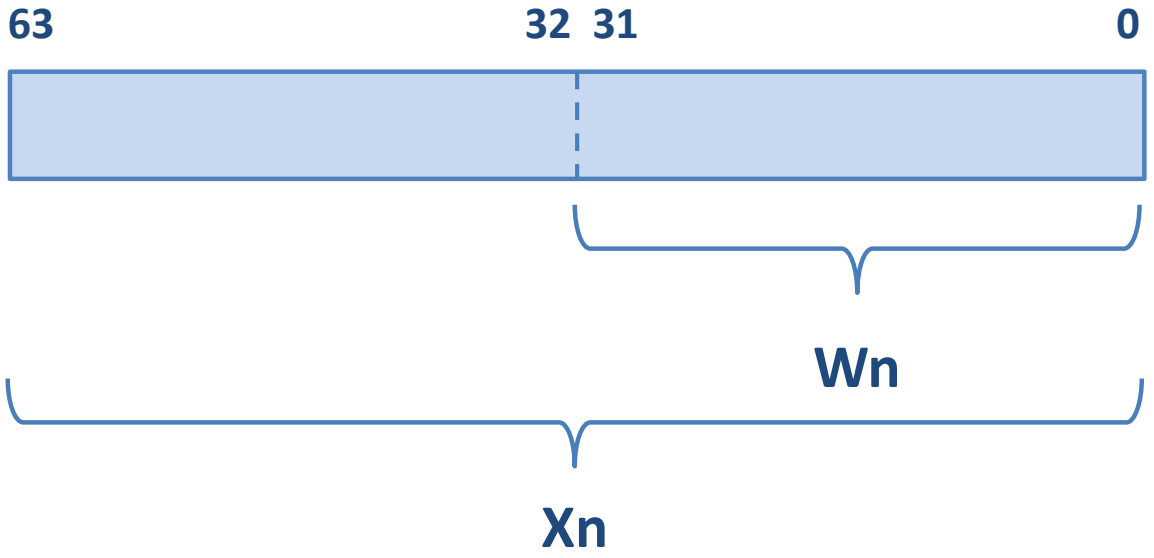
- 64-bit pointers and registers
- Fixed-length 32-bit instructions
- Load/store architecture
- Simple addressing modes
- 32 x 64-bit general-purpose registers (including the R31 the zero/stack register)
- The PC cannot be specified as the destination of a data processing instruction or load instruction.

AArch64 - Registers

In the AArch64 Execution state, each register (X0-X30) is 64-bits wide. The increased width (vs. 32-bit) helps to reduce register pressure in most applications.

Each 64-bit general-purpose register (X0 - X30) also has a 32-bit form (W0 - W30).

Zero register – X31



AArch64 – Load/Store Instructions

LDR – load data from an address into a register.

STR – store data from a register to an address.

```
LDR X0, <addr>  ; load from <addr> into X0
STR X0, <addr>  ; store contents of X0 to <addr>
```

In these cases, X0 is a 64-bit register, so 64-bits will be loaded or stored from/to memory.

AArch64 – Addressing Modes

Base register only: Address to load/store from is a 64-bit base register.

```
LDR X0, [X1]          ; load from address held in X1
STR X0, [X1]          ; store to address held in X1
```

Base plus offset: We can add an immediate or register offset (**register indexed**).

```
LDR X0, [X1, #8]      ; load from address [X1 + 8 bytes]
LDR X0, [X1, #-8]     ; load from address [X1 - 8 bytes]
LDR X0, [X1, X2]      ; load from address [X1 + X2]
LDR X0, [X1, X2, LSL #3] ; left-shift X2 three places
                           before adding to X1
```

•

AArch64 – Addressing Modes

Pre-indexed: source register changed before load

```
LDR W0, [X1, #4]!    ; equivalent to:
                         ADD X1, X1, #4
                         LDR W0, [X1]
```

Post-indexed: source register changed after load

```
LDR W0, [X1], #4     ; equivalent to:
                         LDR W0, [X1]
                         ADD X1, X1, #4
```

AArch64 – Data Processing

- Values in registers can be processed using many different instructions:
  - Arithmetic, logic, data moves, bit field manipulations, shifts, conditional comparisons, etc.
- These instructions always operate between registers, or between a register and an immediate.

Example loop:

```
        MOV X0, #<loop count>
Loop:
        LDR W1, [X2]
        ADD W1, W1, W3
        STR W1, [X2], #4
        SUB X0, X0, #1
        CBNZ X0, loop
```

AArch64 - Branching

`B <offset>`

PC relative branch (+/- 128MB)

`BL <offset>`

Similar to B, but also stores return address in LR (link register), likely a function call

`BR Xm`

Absolute branch to address stored in **Xm**

`BRL Xm`

Similar to BR, but also stores return address in LR

AArch64 - Branching

`RET Xm` or simply `RET`

- Similar to BR, likely a function return
- Uses LR if register is omitted

Subroutine calls:

The Link Register (LR) stores the return address when a subroutine call is made. This is then used at the end of our subroutine to return back to the instruction following our subroutine call.

•

AArch64 – Conditional Execution

The A64 instruction set does not include the concept of widespread predicated or conditional execution (as earlier Arm ISAs did).

The NZCV register holds copies of the N, Z, C, and V condition flags.

A small set of conditional data processing instructions are provided that use the condition flags as an additional input. Only the conditional branch is conditionally executed.

- Conditional branch
- Add/subtract with carry
- Conditional select with increment, negate, or invert
- Conditional compare (set the condition flags)

AArch64 – Conditional Branches

B.cond

Branch to label if condition code evaluates to true, e.g.,

```
CMP X0, #5
B.EQ label
```

CBZ/CBNZ – branch to label if operand register is zero (CBZ) or not equal to zero (CBNZ)

TBZ/TBNZ – branch to label if specific bit in operand register is set (TBZ) or clear (TBNZ)

```
TBZ W0, #20, label ; branch if (W0[20]==#0b0)
```

AArch64- Conditional Operations

`CSEL` – select between two registers based on a condition

```
CSEL X7, X2, X0, EQ ; if (cond==true) X7=X2, else X7=X0
```

There are also variants of this that cause the second source register to be incremented, inverted, or negated.

Instruction: Addition

C Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

- **ADD:** mnemonic – indicates operation to perform
- **b, c:** source operands
- **a:** destination operand

Instruction: Subtraction

Similar to addition - only mnemonic changes

C Code	ARM assembly code
`a = b - c;`	`SUB a, b, c`

- **SUB:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Design Principle 1

Regularity supports design simplicity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Ease of encoding and handling in hardware

Multiple Instructions

More complex code handled by multiple ARM instructions

C Code

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

ARM assembly code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

Design Principle 2

Make the common case fast

- ARM includes only simple, commonly used instructions
- Hardware to decode and execute instructions kept simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions

Design Principle 2

Make the common case fast

- ARM is a **Reduced Instruction Set Computer (RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are **Complex Instruction Set Computers (CISC)**

Operand Location

Physical location in computer

- Registers
- Constants (also called *immediates*)
- Memory

Design Principle 3

Smaller is Faster

- ARM includes only a small number of registers

Instructions with Registers

Revisit ADD instruction

C Code	ARM Assembly Code
	`; X1 = a, X2 = b, X3 = c`
`a = b + c`	`ADD X1, X2, X3`

Operands: Constants\Immediates

- Many instructions can use constants or *immediate* operands
- For example: ADD and SUB
- value is *immediate*ly available from instruction

C Code	ARM Assembly Code
	`; X0 = a, X1 = b`
`a = a + 4;`	`ADD X0, X0, #4`
`b = a - 12;`	`SUB X1, X0, #12`

Generating Constants

Generating small constants using move (MOV):

C Code

```
//int:
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; X0 = a, X1 = b
MOV X0, #23
MOV X1, #0x45
```