arm

Fundamentals of Computer Design

Module 2

© 2021 Arm Limited

Module Syllabus

- Explore a simple processor design.
- Introduce the fundamentals of computer design.
- Outline the principles of instruction set design.
- The Armv8-A Instruction Set Architecture.

© 2021 Arm Limited

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**

© 2021 Arm Limited

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).

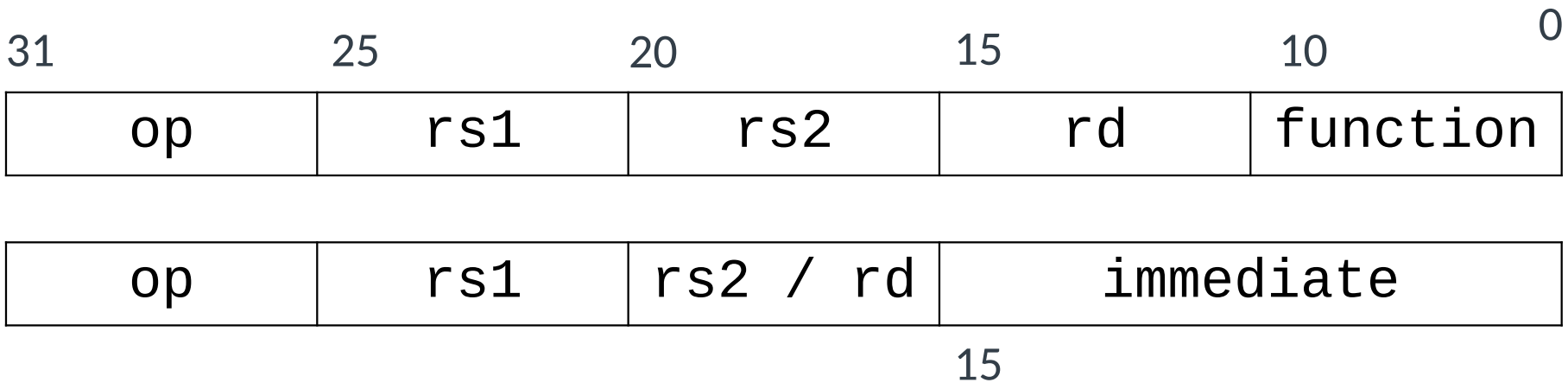
© 2021 Arm Limited

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.:



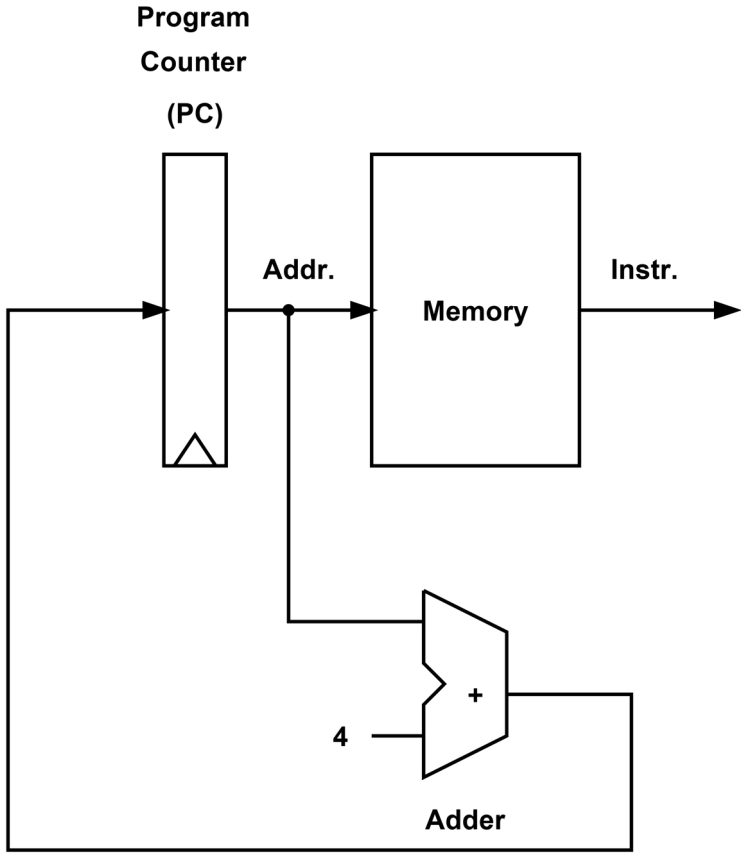
© 2021 Arm Limited
arm

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.

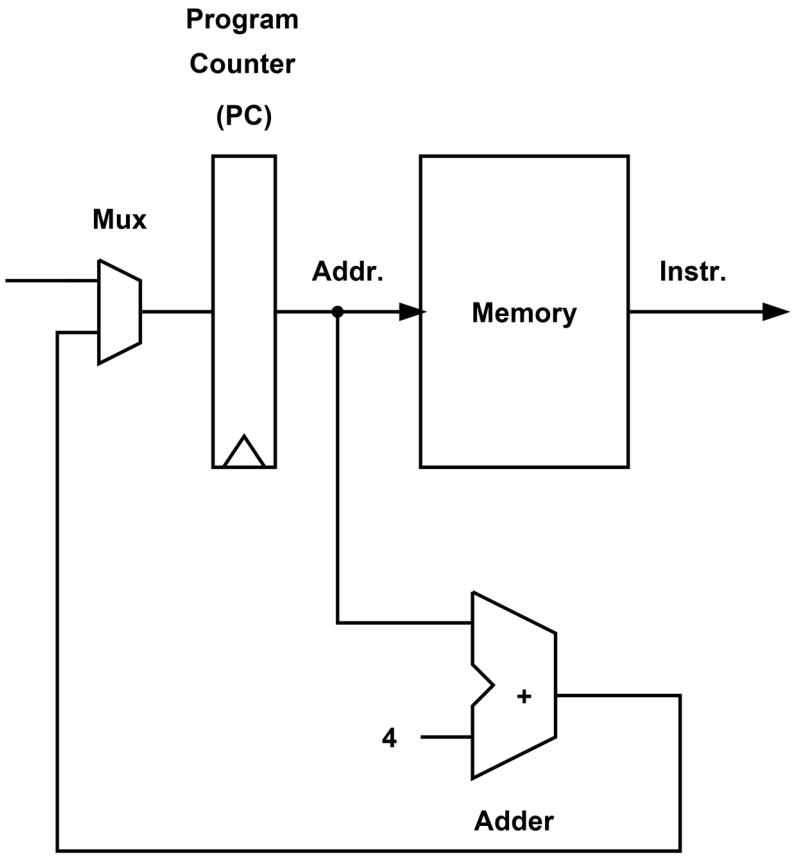
We need to add more logic to actually compute, handle branches, provide registers, access data memory, etc.



© 2021 Arm Limited
arm

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.

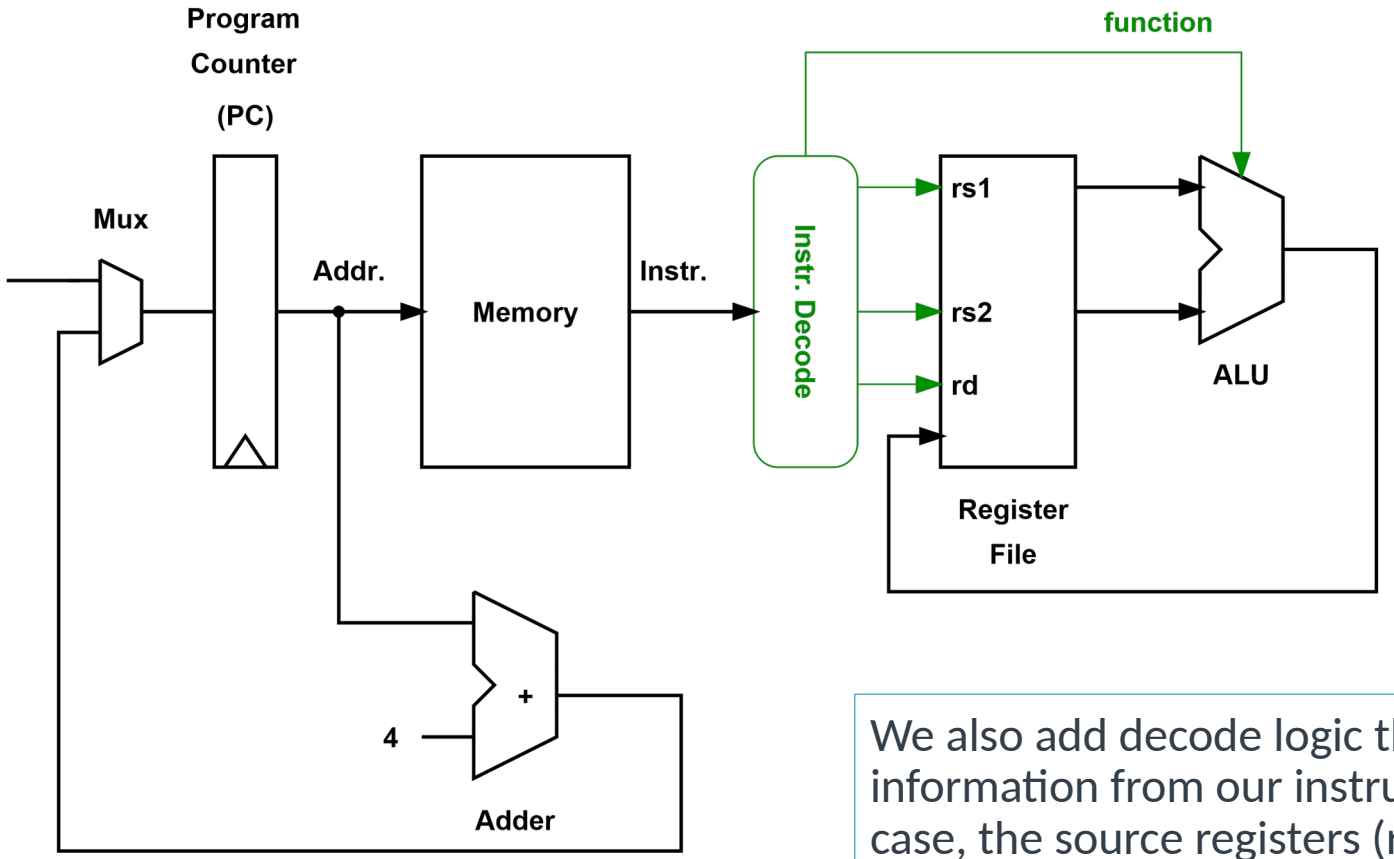


© 2021 Arm Limited

A Processor Datapath – Add a Register File and ALU

We could now execute a simple sequence of ALU operations.

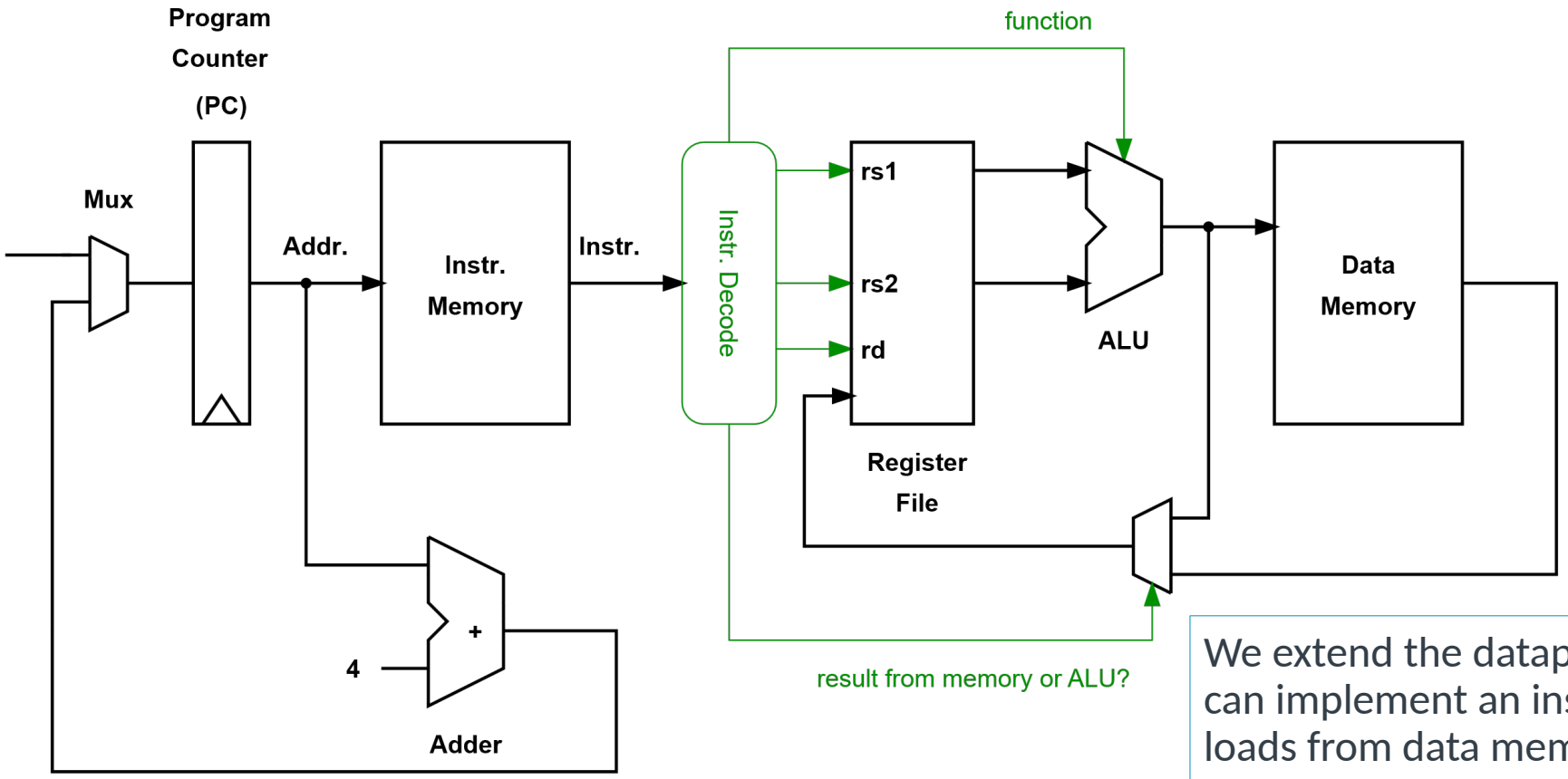
Although, we do not yet have access to data memory and can't branch!



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.

© 2021 Arm Limited
arm

A Processor Datapath – Loading from Data Memory



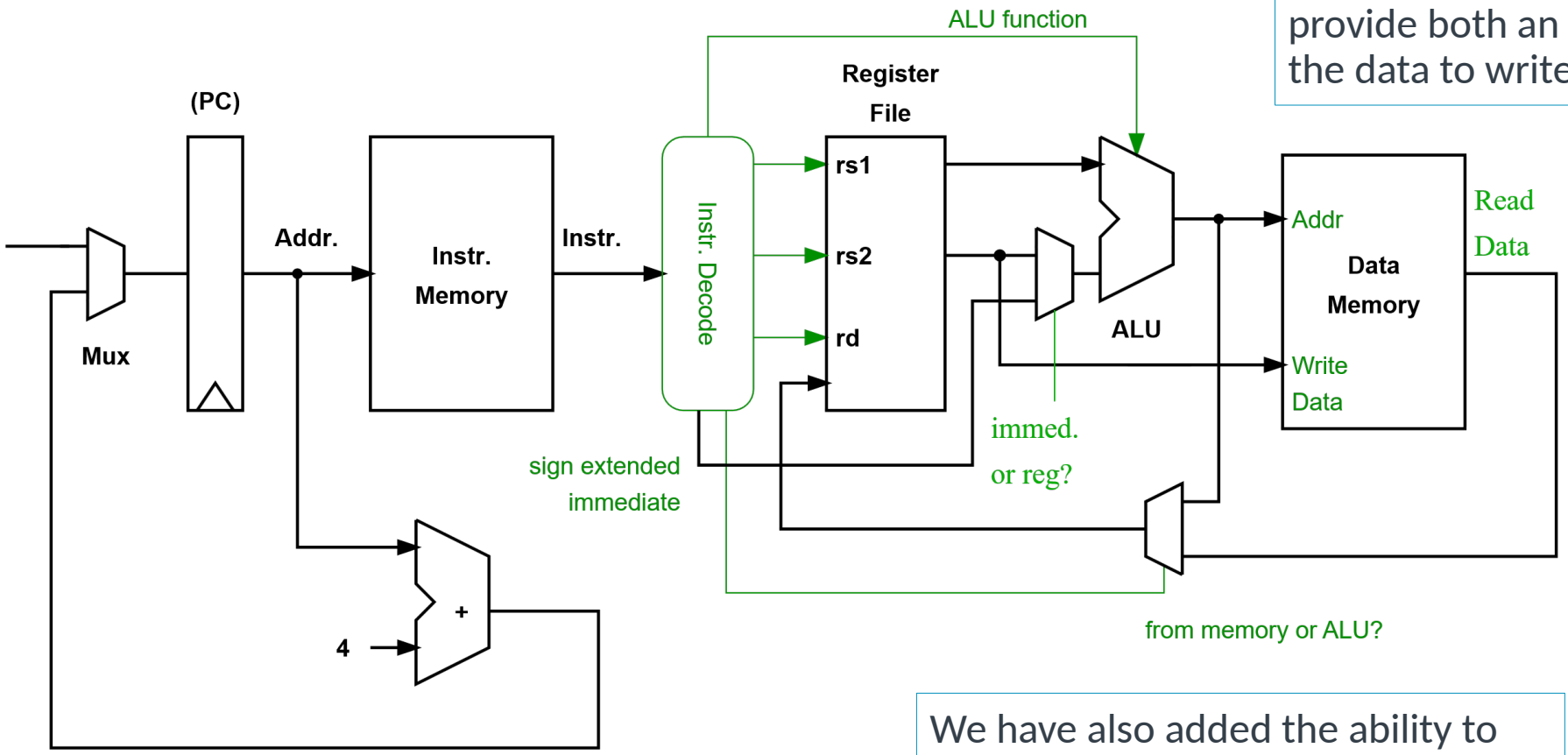
We extend the datapath so we can implement an instruction that loads from data memory.

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

© 2021 Arm Limited

A Processor Datapath – Storing to Data Memory

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

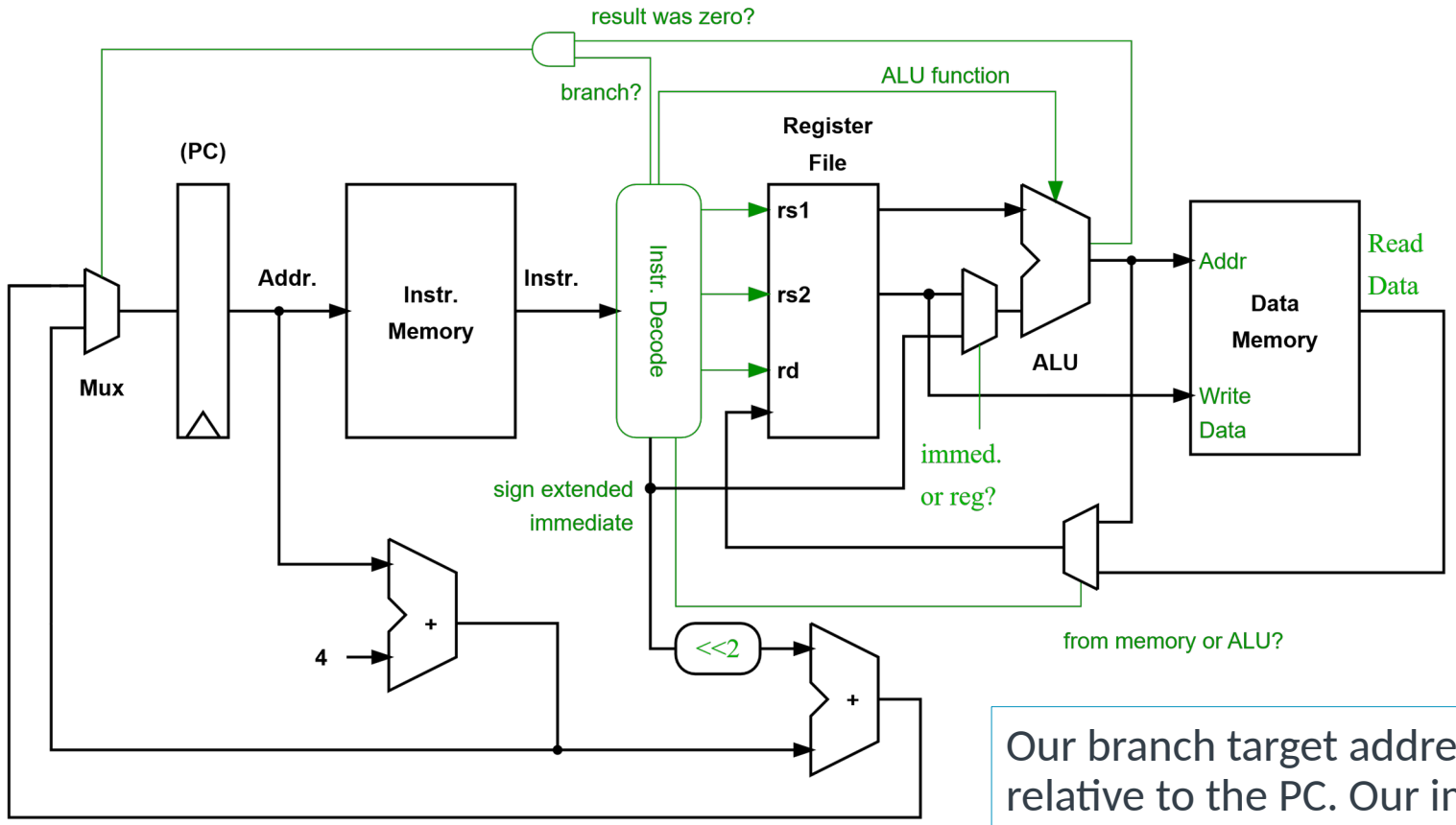


We have also added the ability to extract immediate values from the instructions and use them in ALU operations or address calculations.

© 2021 Arm Limited

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.

© 2021 Arm Limited

The Fundamentals of Computer Design

© 2021 Arm Limited

The Fundamentals of Computer Design

- **Architecture**
  - Set of specifications that allows developers to write software and firmware. These include the instruction set.
- **Microarchitecture**
  - Logical organization of the inner structure of the computer. Sometimes also called the “organization”
- **Hardware or Implementation**
  - The realization or the physical structure, i.e., logic design and chip packaging

© 2021 Arm Limited

Amdahl's Law

- How do we allocate our resources?
- Amdahl's law

$$speedup = \frac{time_{without\_enhancement}}{time_{with\_enhancement}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

- Amdahl's law provides a simple way of calculating the performance gain that can be obtained by improving an element of a computer system.

© 2021 Arm Limited

Amdahl's Law

- Diminishing returns
  - Incremental improvements in speedup gained by enhancing just one portion of our design diminish as improvements are made. Eventually, we reach the limit `1/(1-Fraction_enh).`
  - E.g., If Fraction_enh = 0.5

    Speedup_enh = 2, speedup = 1.33

    Speedup_enh = 4, speedup = 1.6

    Speedup_enh = 10, speedup = 1.8

    Speedup_enh = 100, speedup = 1.98

- If we focus on a single optimization, we will see diminishing returns from investing more hardware. This suggests we should carefully re-evaluate where to make enhancements (and invest hardware) after each optimization is applied.

© 2021 Arm Limited

Complexity

"In engineering, all other things being equal, simpler is always better, and sometimes much better," Robert P. Colwell, from the book "The Pentium Chronicles"

- We normally want to seek simple elegant solutions.
- Complexity adds to design and verification costs.
- We often want to understand what we have built and predict how it will behave.

© 2021 Arm Limited

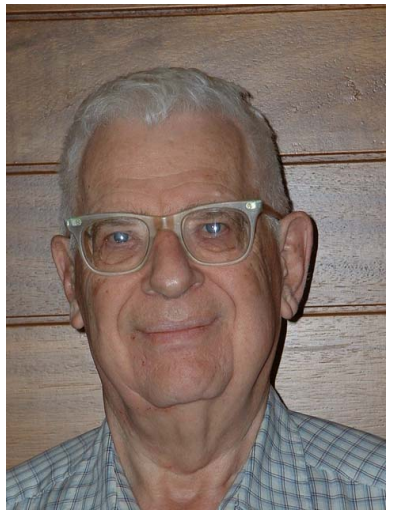
Making the Common-case Fast

- Pitfall: Isn’t every improvement we make worthwhile even if it only provides a small performance improvement?
- No! Enhancements consume design, verification, and implementation resources – they are never free.
- Adding an enhancement may disadvantage the common case:
  - Less time can be spent optimizing the common case.
  - Accelerating complex operations may require the cycle-time to be extended. This will slow all operations.
  - Resources (transistors, power) are redirected to less important (less often used) features.

© 2021 Arm Limited

Making the Common-case Fast

- What tools/techniques do we have at our disposal?
- **Locality**
- **Speculation**
- **Prediction**
- **Indirection***
- **Parallelism**
- **Specialization**



* "All problems in computer science can be solved by another level of indirection" – David Wheeler (1927-2004), University of Cambridge

© 2021 Arm Limited

Benchmarks

- We use benchmarks to determine what the common cases are and to help guide the design process.
- The challenge is to always carefully evaluate ideas quantitatively. This is difficult as the actual workload is often unknown and the design space of possible designs is extremely large.
- There is also the risk of looking backward:

> “Computer architects often err by preparing for yesterday’s computations,”
>
> Prof. Bill Dally, Stanford and Chief Scientist Nvidia.

© 2021 Arm Limited

The Instruction Set Architecture

© 2021 Arm Limited

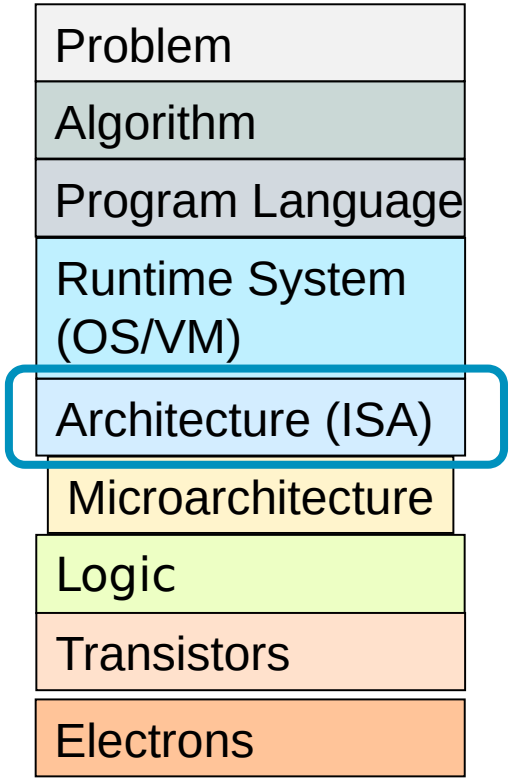
Instruction Set Architecture (ISA)

- 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



© 2021 Arm Limited
arm

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?

© 2021 Arm Limited

Instruction Set Architecture

High-level interface

Low-level interface

- Execute high-level languages directly.
- Execute complex instructions (CISC).
- Tailor instruction set for pipelined and high-performance implementations. Expose the instruction pipeline to the compiler so it can optimize code and help simplify the hardware (RISC).
  We will explore this approach.
- Provide additional explicit information about the dependencies between instructions. E.g., VLIW or
- Specify individual data transfers, e.g., Transport Triggered Architectures (TTA)

© 2021 Arm Limited

Instruction Set Architecture

- The best instruction set is the one that yields the “best” implementation.
- Changing the instruction set is difficult and happens infrequently .
- The factors that influence instruction set design do change over time, e.g., applications, programming languages, compiler technology, transistor budgets, and the underlying fabrication technology.
- We need to take care not to include “features” that will be regretted later.

© 2021 Arm Limited

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.

© 2021 Arm Limited

Instruction Set Architecture

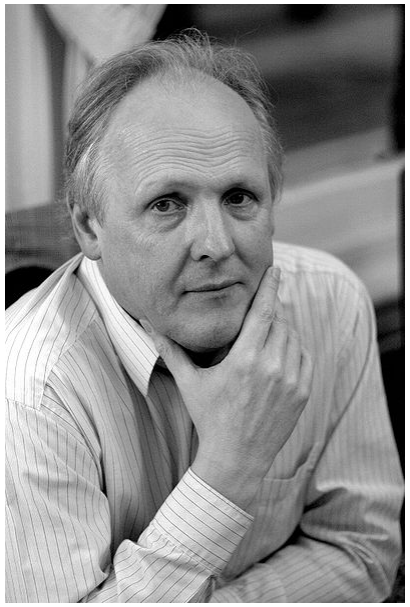
Common features of RISC instruction sets:

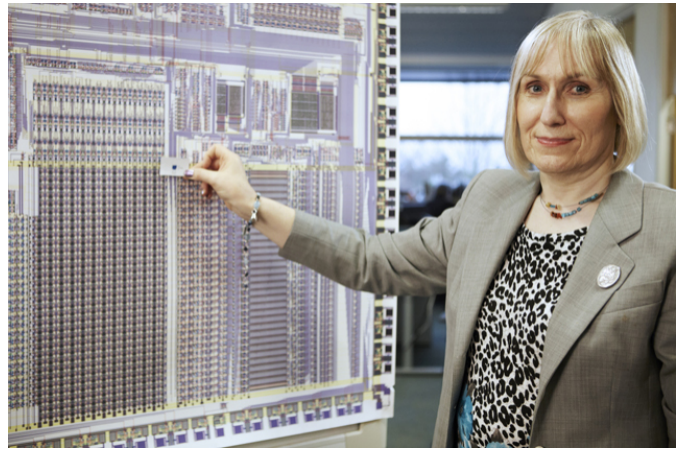
- Fixed length instruction encodings (or a small number of easily decoded formats)
- Each instruction follows similar steps when being executed.
- Access to data memory is restricted to special load/store instructions (a so-called **load/store architecture**).

© 2021 Arm Limited

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

© 2021 Arm Limited

CISC

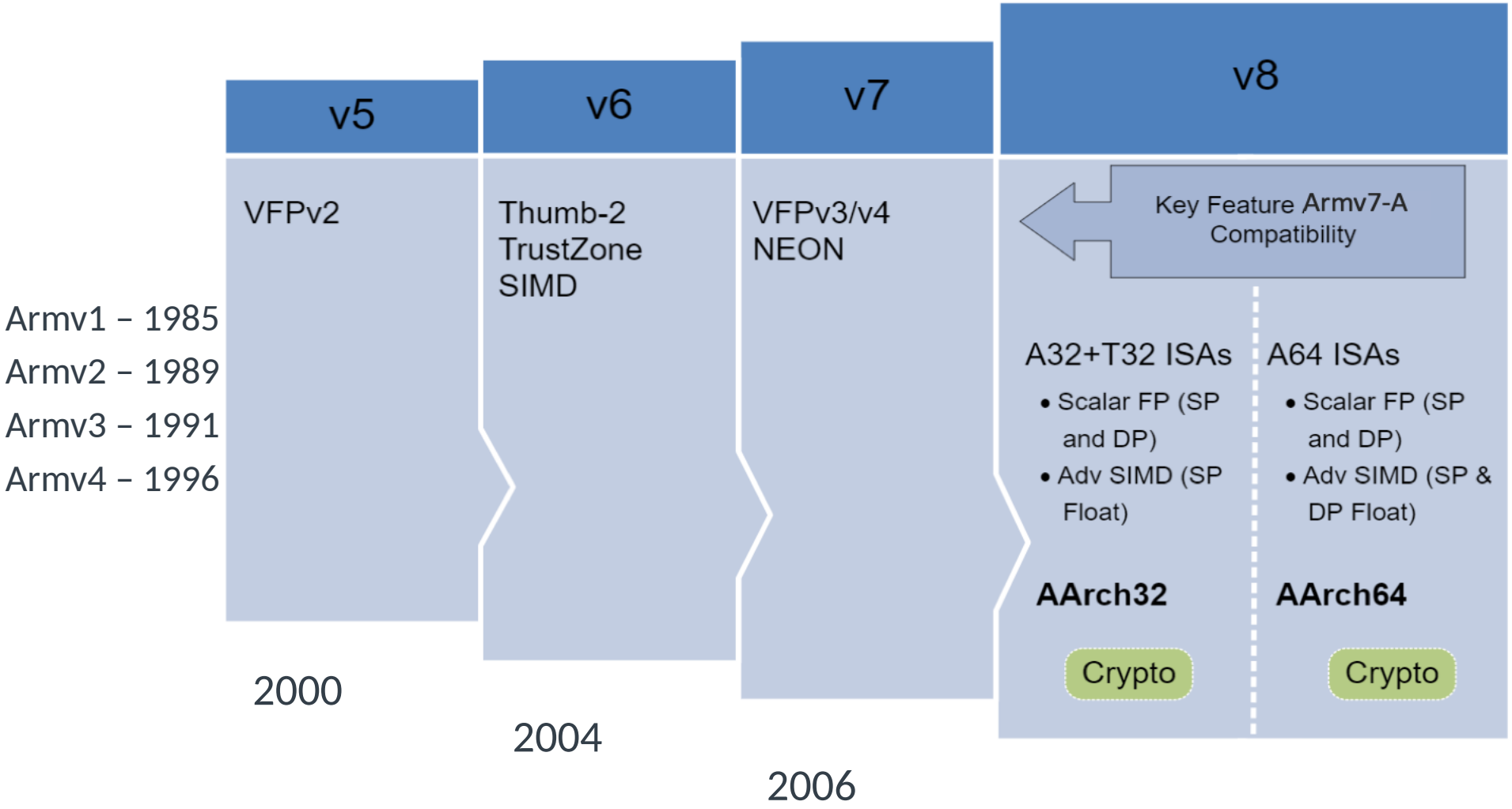
- Not all processors use RISC ISAs; some are Complex Instruction Set Computers (CISC).
- CISC designs have evolved since the 1970s.
- How are modern CISC machines (e.g., x86) implemented?
  - They first convert the CISC instructions to (possibly numerous) RISC-like micro-ops!
  - Their microarchitectures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

© 2021 Arm Limited

The Armv8-A ISA

© 2021 Arm Limited

Case Study: The Armv8 Architecture



© 2021 Arm Limited
arm

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.

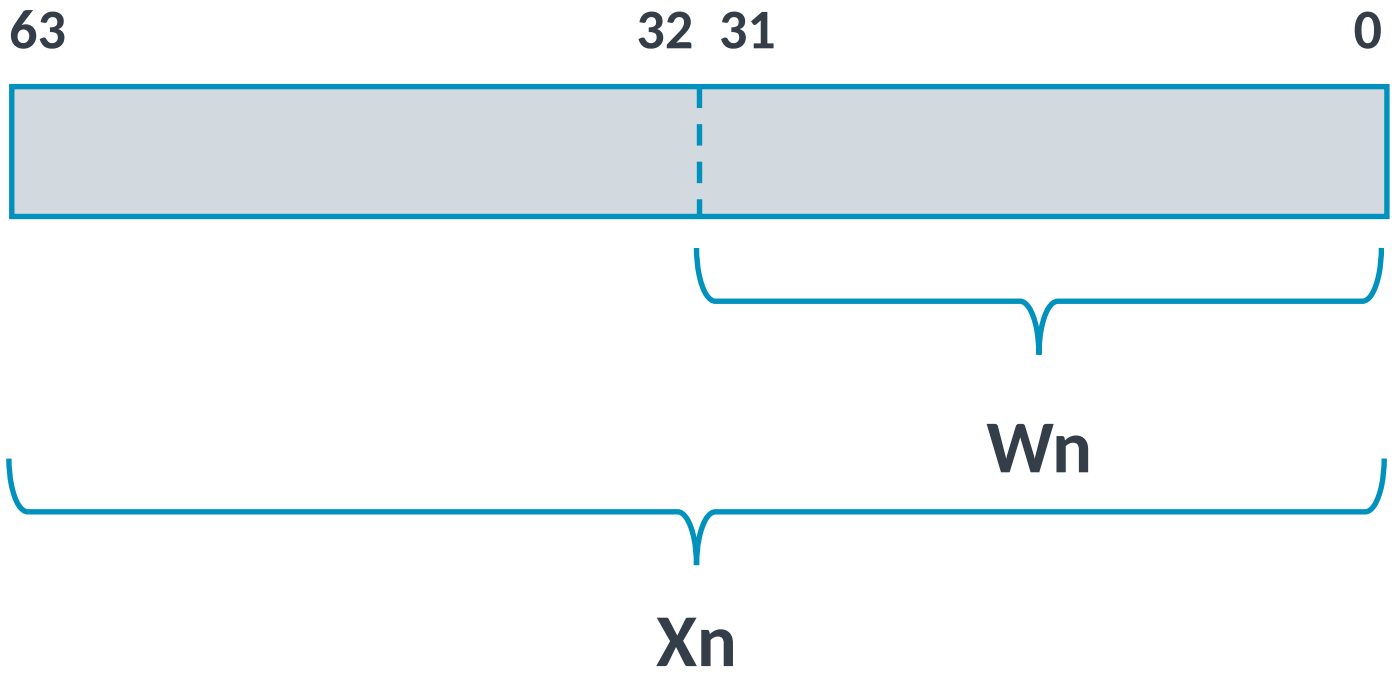
© 2021 Arm Limited

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



© 2021 Arm Limited
arm

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.

© 2021 Arm Limited

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
```

•

© 2021 Arm Limited

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
```


© 2021 Arm Limited

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
```


© 2021 Arm Limited

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

© 2021 Arm Limited

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.

© 2021 Arm Limited

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)

© 2021 Arm Limited

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)
```


© 2021 Arm Limited

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.

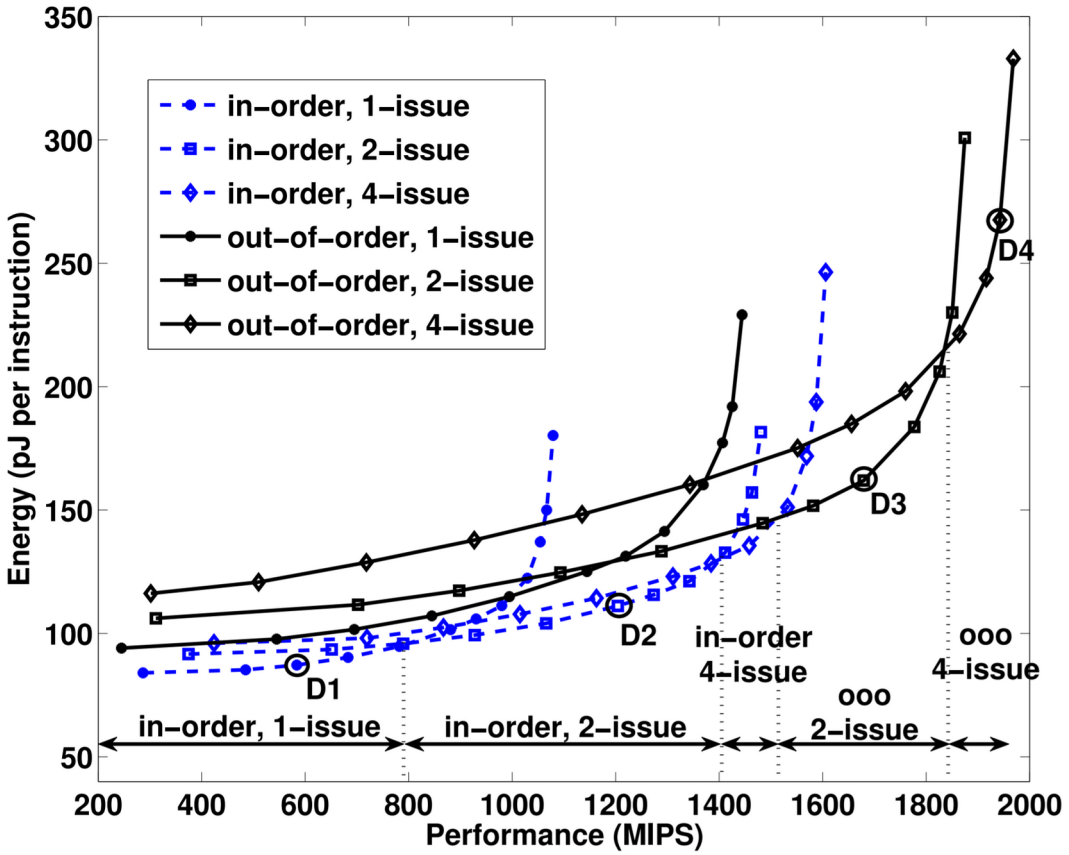
© 2021 Arm Limited

Backup Slides

© 2021 Arm Limited

The Fundamentals of Computer Design

- Let's assume our architecture and fabrication technology are fixed.
- There are still many different microarchitectural design choices.
- If we want to minimize power consumption, the "best" microarchitecture will depend on the required performance.



O. Azizi, A. Omid & A. Mahesri, M. Aqeel & B. Lee, L. Benjamin & S. Patel, P. Sanjay & M. Horowitz. *Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis*, 2010. ACM SIGARCH Computer Architecture News. 38. 26-36. 10.1145/1815961.1815967.

© 2021 Arm Limited

AArch64 – How Does It Differ from Older Arm ISAs?

- Conditional execution mostly dropped
- No free shifts in arithmetic instructions
- Program counter not a part of integer register set
- No load/store multiple instructions
- Adopts a more regular instruction encoding

© 2021 Arm Limited

Compressed Instruction Sets

- We could encode instructions using 16-bits to produce much smaller programs.
- This improved code density would come at the cost of some performance.
- The T32 (Thumb-2) instruction set (part of the Armv8 AArch32 execution state) allows us to mix 16- and some 32-bit instructions (without a mode change).

This allows us to improve code density and maintain performance, e.g., in performance critical loops.

© 2021 Arm Limited

Specifying More Work in One Instruction

- It is often beneficial to specify more work in a single instruction if this simplifies our hardware or enables us to make better use of it.
  - See earlier comments regarding Armv8 addressing modes.
- Going further, *we* might be able to specify more work and hence fetch fewer instructions, or perform more work per cycle.
- We might actually also be able to reduce fetch and control overheads in this way, *e.g.*, SIMD or vector instructions.

© 2021 Arm Limited

Early RISC Ideas: IBM 801 (~1974-1980)

The IBM 801's architectural aims to:

- Make effective use of the registers and ensure registers are general-purpose.
- Avoid complex instructions where the same effects can be realized by simple ones. Ensure hard-wired control is possible.
- Use separate instruction and data caches
- Ensure all instructions are usable by compilers.
- Provide an optimizing compiler.

"In some sense, the 801 appears to be rushing in the opposite direction to the conventional wisdom of this field. Namely, everyone else is busily moving software into hardware and we are clearly moving hardware into software. Rather than consuming the projected cheaper, faster hardware, we are engaged in an effort to save circuits, cut path lengths and reduce functions at every level of the normal hierarchy."

From "The 801 Microcomputer – An Overview", Internal IBM memo, 1976.

© 2021 Arm Limited