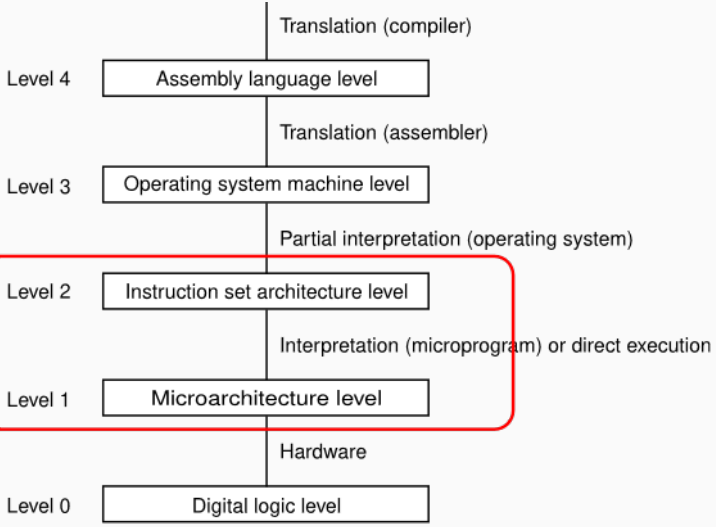
Instruction Set Architectures and the Data Path

Peyman Afshani and Jean Pichon-Pharabod

2024

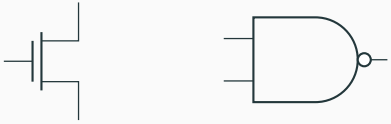
Department of Computer Science – Aarhus University

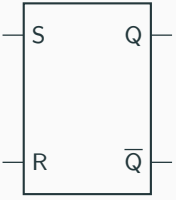
Multi-level system



Goal

Last week, we saw how transistors are used to implement basic gates and circuits, including basic memory.





This week, we will look at:

- How to design an Instruction Set Architecture
- How to implement such instructions in hardware

Design

Design of a hardware architecture

Definition

Hardware architecture = Instruction Set Architecture (ISA) + memory consistency model

Design of a hardware architecture

Definition

Hardware architecture = Instruction Set Architecture (ISA) + memory consistency model

- The ISA defines the instructions that software can use to control the processor:
  - Syntax:
    - Instruction format
    - Operands
    - Addressing modes for registers and memory
  - Semantics: effects on registers, memory, ...

Design of a hardware architecture

Definition

Hardware architecture = Instruction Set Architecture (ISA) + memory consistency model

- The ISA defines the instructions that software can use to control the processor:
  - Syntax:
    - Instruction format
    - Operands
    - Addressing modes for registers and memory
  - Semantics: effects on registers, memory, ...
- The memory consistency model defines how the effects of instructions interact:
  - what value is read from an address when there are multiple writes to it
  - ...

Instruction Set Architecture (ISA) design

There are many considerations behind an ISA design:

- What is the **purpose** of the machine being designed?
- Which **operations** and data **types** must be supported?
- How should **memory** be addressed?
- What are the **efficiency** constraints: execution time, area, power?
- What are the **market** constraints: price/fit, compatibility, developer support?
- What are the **legacy** constraints: backwards compatibility?

Instruction Set Architecture (ISA) design

There are many considerations behind an ISA design:

- What is the **purpose** of the machine being designed?
- Which **operations** and data **types** must be supported?
- How should **memory** be addressed?
- What are the **efficiency** constraints: execution time, area, power?
- What are the **market** constraints: price/fit, compatibility, developer support?
- What are the **legacy** constraints: backwards compatibility?

Many of these considerations are in tension:

- A **minimum** instruction set is *sufficient* but not *convenient*.
- A **large** one is *convenient*, but *inefficient* to support.

$\rightsquigarrow$ so ISA design is about trade-offs.

Instruction Set Architecture (ISA) design

1. **Functionality**: what the instructions provide
   - Arithmetic: integer, floating point
   - Logic: bit manipulation and testing
   - Control: branching, function call, ...
   - Other:
     - graphics
     - networking
     - cryptography (e.g. AES instruction set)
     - DSP (digital signal processing)
     - data conversion & compression
     - machine learning
     - ...
2. **Format**: representation for each instruction
3. **Semantics**: effect of the instruction

Instruction format

Instruction format

Instructions are represented as sequences of bits in memory (usually multiples of bytes):

- *Opcode*: specifies operation to be performed
- *Operands*: specify data values on which to operate
- *Result location*: specifies where result is to be placed

Instruction format

There are also trade-offs about instruction length:

1. *Fixed-length* (MIPS, aarch64, RISC-V, ...)
   - Every instruction has the same size
   + Hardware for decoding is less complex, and thus can run faster
   – Wastes space, since some instructions do not use all bits

Instruction format

There are also trade-offs about instruction length:

1. *Fixed-length* (MIPS, aarch64, RISC-V, ...)
   - Every instruction has the same size
   + Hardware for decoding is less complex, and thus can run faster
   – Wastes space, since some instructions do not use all bits

2. *Variable-length* (x86, Arm Thumb-2, ...)
   - Some instructions are shorter/longer than others
   – Decoding is more complex
   + Allows instructions with **variable** number of operands (from zero to many)
   + Optimal use of memory

Registers

Words

A hardware architecture typically works with a given **word** size
$\rightsquigarrow$ a register contains a word, a bus can transfer a word, ...
(reality is a bit more complicated)

Modern processors have word sizes typically in the range between 8 bits and 64 bits.

This has implications for:

- **registers**: what data they can hold at once
- **memory access**: how much of memory can be directly accessed:
  an 8-bit processor can directly address 256 words; vs a 64-bit processor, 2^{64} words.

General-purpose registers (GPRs)

Registers are the most common operands to instructions:

- **High-speed** storage mechanism, part of the processor (stored on chip)
- Temporary storage during computation, operand for arithmetic operation
- Each GPR holds an **integer** (which can be interpreted as an **address**)

Note: Some processors require all operands for an arithmetic operation to come from general-purpose registers

Remark: Floating-point registers are typically separate

General-purpose registers (GPRs)

Consider the **task** below:

- Start with variables X and Y in memory
- Add X and Y and place the result in variable Z (also in memory)

Example steps, assuming registers 1-3 are **available**:

1. Load a copy of X into register 1
2. Load a copy of Y into register 2
3. Add the value in register 1 to the value in register 2, and put the result in register 3
4. Store a copy of the value in register 3 in Z

Registers and memory

Registers are a scarce resource
$\rightsquigarrow$ need to do careful **register allocation**:
which values should be kept in registers at each point?
Performed by the programmer, or automatically by the compiler.

What to do when a register is **needed** for a computation, but all registers are used?
$\rightsquigarrow$ **spill a register**: save current contents of a register in memory
... then reload it from memory when the value is needed.

See y3 Compilers.

Dealing with large values

Some operations have results (or operands!) that don't fit in a single register:

- by "accident": the multiplication of two n-bit integers does not always fit in n bits... but in $2n$ bits
- on purpose: advantageous to group operations often performed together (vector, ...)

Several approaches:

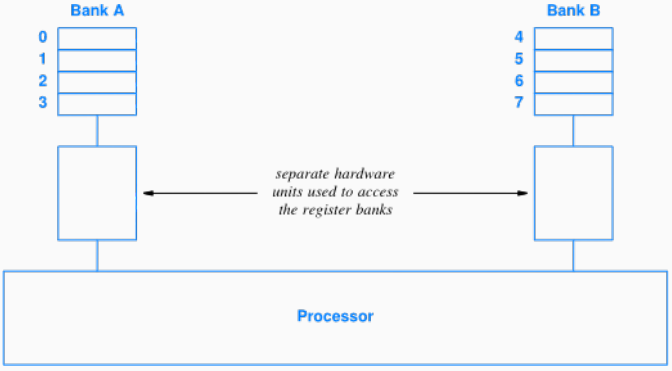
- Uses a **contiguous** pair of registers (either implicitly defined or programmer-specified)
- Use **wide** registers, for example 128 bits
- ...

Register banks

In some architectures, registers are **partitioned** into disjoint sets called *banks*[1].

Register banks

In some architectures, registers are **partitioned** into disjoint sets called *banks*[1].
The ALU instruction takes at most one operand from each *bank* (runtime error otherwise).



It is a hardware detail that optimises performance, but **complicates** programming.

[1]This term is also used for when different exception levels have their own version of certain registers.

Register bank conflicts

Even trivial programs can cause *register conflicts*.

Example: How to lay out this sketch program using two register banks?

$$
\begin{aligned}
a &:= \text{add } x\ y \\
b &:= \text{sub } x\ z \\
c &:= \text{add } y\ z
\end{aligned}
$$

No in-register solution!

Assembly, CISC, and RISC

Assembly

An **assembly** is a programming language in very close correspondence with machine language. Features may include **symbolic labels**, comments, macros (compile-time expanded), etc.

```
0000000100003f78        sub     sp, sp, #0x10
0000000100003f7c        mov     w8, #0x1
0000000100003f80        str     w8, [sp, #0x8]
0000000100003f84        ldr     w8, [sp, #0x8]
0000000100003f88        add     w8, w8, #0x1
0000000100003f8c        str     w8, [sp, #0x8]
0000000100003f90        subs    w8, w8, #0xc
0000000100003f94        cmp     w8, wzr
0000000100003f98        b.ne    0x100003f84
0000000100003f9c        ldr     w0, [sp, #0x8]
0000000100003fa0        add     sp, sp, #0x10
0000000100003fa4        ret
```

Given by `otool -tvV a.out`

```
        .section        __TEXT,__text
        .globl _main
.macro incloc
        ldr         $1, [sp, $0]
        add         $1, $1, #0x1
        str         $1, [sp, $0]
.endm
_main:
        sub         sp, sp, #16 ; init
        mov         w8, #1
        str         w8, [sp, #8]
L:
        incloc #8, w8 ; increment
        subs        w8, w8, #12 ; compare
        cmp         w8, wzr
        b.ne        L
        ldr         w0, [sp, #8] ; return
        add         sp, sp, #16
        ret
```

A compiler from assembly to machine code is called an **assembler**.

A compiler from machine code to (basic) assembly is called a disassembler.

Complex Instruction Set Computers (CISC)

Early ISAs were designed for **human** programmers writing assembly (IBM before Power, x86, ...)

- **Powerful** instructions were frequently added to optimise **productivity** (and speed): mixed computation & memory access, memory-to-memory, ...
  Typical CISC instruction:

$$
\begin{aligned}
\texttt{mult}\ r_1,\ r_2,\ r_3 = \quad & \text{read from register } r_1, \text{ call the result } x \\
& \text{read from register } r_2, \text{ call the result } y \\
& \text{read from register } r_3, \text{ call the result } z \\
& \text{read from address } x, \text{ call the result } v_1 \\
& \text{read from address } y, \text{ call the result } v_2 \\
& \text{multiply } v_1 \text{ and } v_2, \text{ call the result } v_3 \\
& \text{write to address } z \text{ value } v_3 \\
& \text{increment } pc
\end{aligned}
$$

- Performance came from reducing number of **fetch-decode-execute cycles**

The motivation for RISC

Observations: late 70s–early 80s

- The complexity of instructions **greatly increased** complexity of processors!
- ... yet most programs used a tiny minority of the available instructions!
- ... and optimising compilers have matured, so they can efficiently deal with the tedium

  (with very limited memory, early compilers were very constrained, and would often generate slow, large code)

Idea Instead, use simple instructions working on many registers
$\rightsquigarrow$ much **simpler** $\rightsquigarrow$ **smaller** $\rightsquigarrow$ **faster** hardware

Reduced Instruction Set Computers (RISC)

Slogan: rather four simple instructions than one complex.

Typical RISC instructions:

`ldr` r_1, $[r_2]$ =
- read from register r_2, call the result x
- read from address x, call the result v
- write to register r_1 value v
- increment pc

`mul` r_1, r_2, r_3 =
- read from register r_2, call the result v_2
- read from register r_3, call the result v_3
- multiply v_2 and v_3, call the result v
- write to register r_1 value v
- increment pc

Program:

```
ldr r4, [r2]
ldr r5, [r3]
mul r6, r4, r5
str r6, [r1]
```

The RISC revolution

"A RISC processor has an instruction set that is designed for efficient execution by a pipelined processor and for code generation by an optimizing compiler." — Michael Slater, Microprocessor Report

- Having many registers make the job of the compilers easier
- Memory addressing is handled explicitly via `load` and `store` instructions
  $\rightsquigarrow$ "load-store architecture"
- All calculation is performed on registers, frequently during memory wait cycles
- Instruction format becomes **uniform** $\rightsquigarrow$ simpler decoding $\rightsquigarrow$ simpler hardware
- Tedious to program, but easier to execute & **pipeline** (see next)
  $\rightsquigarrow$ higher clock frequencies

It worked **really well**! Berkeley RISC, (Stanford) MIPS, Arm, Power(PC), RISC-V, ...

Can we go further? — see later

Convergence of CISC and RISC

Convergence 1: Many modern 'CISC' CPUs (like Intel's Core processor) are **internally** a highly efficient RISC processor, and CISC instructions are broken into short RISC-like instructions.

Convergence of CISC and RISC

Convergence 1: Many modern 'CISC' CPUs (like Intel's Core processor) are **internally** a highly efficient RISC processor, and CISC instructions are broken into short RISC-like instructions.

Convergence 2: Modern 'RISC' CPUs (ARM, SPARC and POWER, ...) have lots of instructions and a sophisticated internal architecture, adopting some 'CISC' features
$\rightsquigarrow$ *Rationalised* Instruction Set Computers

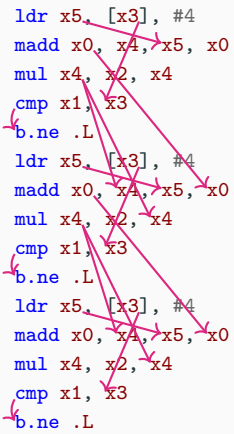
Instruction pipeline

Motivation

Observation 1 In most programs, many instructions do not depend on each other.

```
// polynomial evaluation
for (int i = 0; i < deg; i++) {
  r += p[i] * xn;
  xn *= x;
}
```

```
; x0: r, x1: &p[deg], x2: x
; x3: &p[i], x4: xn, x5: p[i]
L:  ldr x5, [r3], #4 ; x5 := p[i], i++
    madd x0, x4, x5, r0; r += r5 * xn
    mul x4, x2, x4 ; xn *= x
    cmp x1, x3 ; i < deg
    b.ne .L
```



Dense, but not totally ordered

credits: Brian Railing & Nathan Beckmann

Motivation

Recall the fetch-decode-execute cycle, with the 'classic RISC' steps highlighted:

1. **F**etch the next **i**nstruction IF
2. **D**ecode the **i**nstruction ID
3. Execute:
   - **Ex**ecute the arithmetic operation specified by the opcode EX
   - **Mem**ory read or write, if needed MEM
   - **W**rite**b**ack result to registers WB

Motivation

Recall the fetch-decode-execute cycle, with the 'classic RISC' steps highlighted:

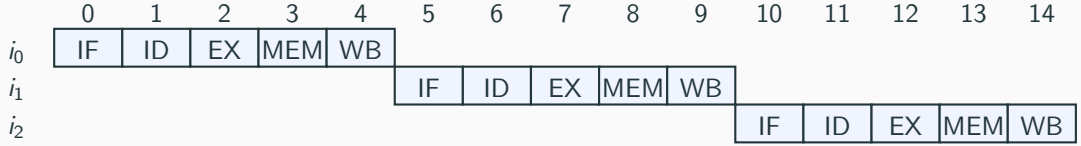
1. **F**etch the next **i**nstruction IF
2. **D**ecode the **i**nstruction ID
3. Execute:
   - **Ex**ecute the arithmetic operation specified by the opcode EX
   - **Mem**ory read or write, if needed MEM
   - **W**rite**b**ack result to registers WB

Observation 2 Different stages of an instruction (largely) use different hardware components ... so **why make them wait?**

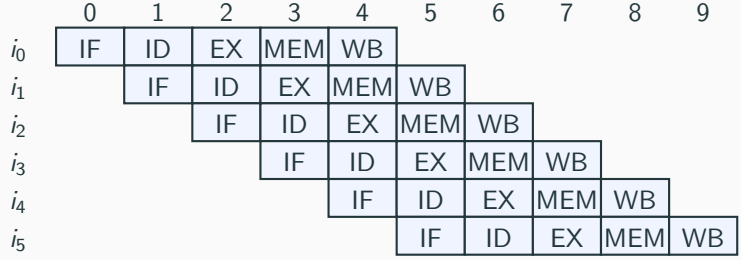
⇝ Exploit the **Instruction-Level Parallelism!**

Instruction pipeline

Instead of executing instructions one by one...

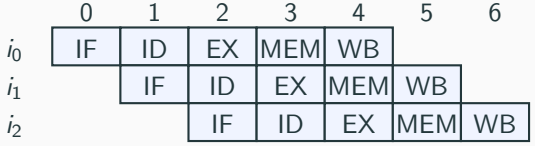


...have a **pipeline** of partially executed instructions:

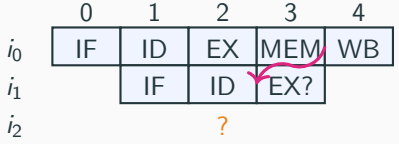


Idea vs. non-ideal pipelining

Ideal case: independent instructions:



Example non-ideal case: if there is a dependency (data hazard):



Need to heed **hazards** (or bite the bullet)

Pipeline hazards

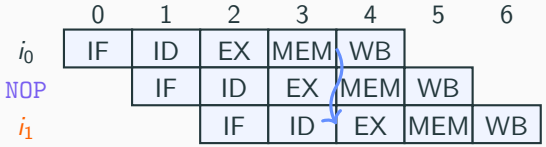
Types:

- Structural hazard: contention for hardware
- Data hazard: one instruction depends on the results of another
- Control hazard: what to execute after a branch?

Pipeline hazards

The burden can be

- on the programmer (for example early MIPS[1], DSPs): for example insert `NOP`s

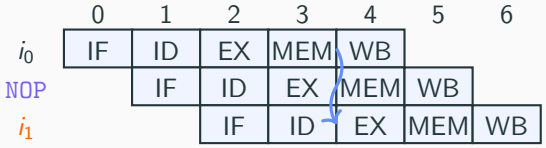


[1] Used to mean 'Microprocessor without Interlocked Pipelined Stages'

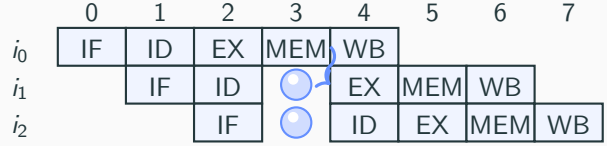
Pipeline hazards

The burden can be

- on the programmer (for example early MIPS[1], DSPs): for example insert `NOP`s



- on the hardware (usual): **stall** $\rightsquigarrow$ bubble

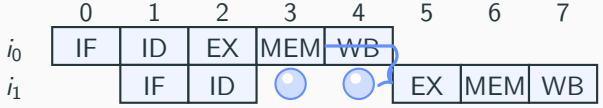


[1] Used to mean 'Microprocessor without Interlocked Pipelined Stages'

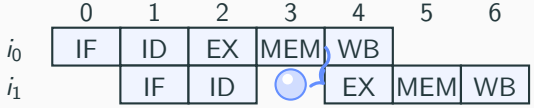
Optimisation: Forwarding

In case of a data hazard...

...instead of waiting for the end of the previous instruction...



...have some mechanism to **forward** the value as soon as it is available:



Optimisation: Branch prediction

What should the pipeline do after a branch?

- Approach 1: do nothing special
  The machine executes the instructions after a branch until the next *pc* is determined
  $\rightsquigarrow$ branch delay slots

Optimisation: Branch prediction

What should the pipeline do after a branch?

- Approach 1: do nothing special
  The machine executes the instructions after a branch until the next *pc* is determined
  $\rightsquigarrow$ branch delay slots

- Approach 2: stall
  ...but $\sim$ 20% of instructions are branches (VERY approximate, completely depends on workload)
  $\rightsquigarrow$ the pipeline would stall all the time!

Optimisation: Branch prediction

What should the pipeline do after a branch?

- Approach 1: do nothing special
  The machine executes the instructions after a branch until the next *pc* is determined
  $\rightsquigarrow$ branch delay slots

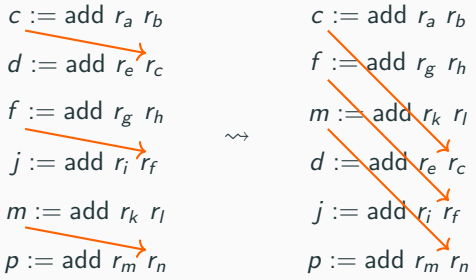
- Approach 2: stall
  ...but $\sim$ 20% of instructions are branches (VERY approximate, completely depends on workload)
  $\rightsquigarrow$ the pipeline would stall all the time!
  $\rightsquigarrow$ **branch prediction**:
  - guess which side is taken
  - execute speculatively
  - roll back if needed

  Works *really well* in practice!
  (+see trick on next slide)

Pipelining for the programmer

For performance, programs should accommodate the instruction pipeline to minimise stalls:

- Structural hazard: schedule to avoid contention
- Data hazards: delay references to result register(s)



- Control hazards: avoid introducing unnecessary branches
  useful trick: compute both branches, then pick the result with a conditional move:

$$\text{if } C \text{ then } X \text{ else } Y \quad = \quad C * X + \overline{C} * Y \quad \text{(terms and conditions apply)}$$

Very long Instruction Word (VLIW) architectures

Idea: go one step further than RISC:

- allow programs to explicitly specify instructions to execute in parallel
- put the burden of scheduling and avoiding hazards on the programmer (like early MIPS)

Works quite well for specific DSP! For example, scalar product on a C6 (credit: Daniel Étiemble):

Unit/cy	0	1	2	3	4	5	6	7
.D1	LDW							
.D2	LDW							
.M1						MPY		
.M2						MPYH		
.L1								ADD
.L2								ADD
.S1		SUB						
.S2			B					

Unit/cy	0	1	2	3	4	5	6	7
.D1	LDW	LDW1	LDW2	LDW3	LDW4	LDW5	LDW6	LDW7
.D2	LDW	LDW1	LDW2	LDW3	LDW4	LDW5	LDW6	LDW7
.M1						MPY	MPY1	MPY2
.M2						MPYH	MPYH1	MPYH2
.L1								ADD
.L2								ADD
.S1		SUB	SUB1	SUB2	SUB3	SUB4	SUB5	SUB6
.S2			B	B1	B2	B3	B4	B5

Itanium

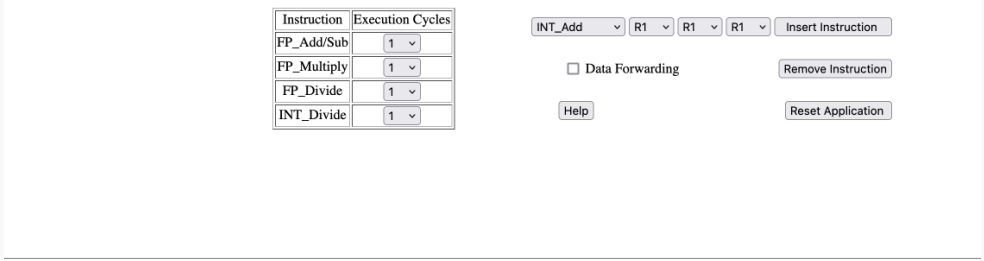
Idea: Explicitly Parallel Instruction Computing (EPIC):
do the same, but general-purpose
$\rightsquigarrow$ push all the instruction scheduling/hazard handling/etc. onto the compiler
$\rightsquigarrow$ "RISC squared"

Didn't live up to the expectations:

- hard to write a compiler that **statically** beats the hardware's **dynamic** (runtime) decisions
- implementation problems
- ...

Simulator

Pipeline simulator:
`http://www.ecs.umass.edu/ece/koren/architecture/windlx/main.html`



Branching

Handling branches

1. Absolute branch: `jump 0x05DE`
   - Typically named `jump`, with a single-operand being an address
   - Assigns operand value to internal register *PC*
2. Relative branch: `br +8`
   - Typically named `br`, with a single-operand being a signed value
   - Adds operand to internal register *PC*

Handling conditional branches

- branch with register condition (e.g. MIPS):

```
        bne $t4, $t5, 1000   # comparison specified inside branch
        nop                  # MIPS needs branch delay slots
        move $t3, 0
...
1000: move $t3, 1
```

Handling conditional branches

- branch with register condition (e.g. MIPS):

```
        bne $t4, $t5, 1000   # comparison specified inside branch
        nop                  # MIPS needs branch delay slots
        move $t3, 0
...
1000: move $t3, 1
```

- flag registers (e.g. aarch64):
  - include special registers known as **flags**: zero, carry, overflow, ...
    `https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/Processor-state`
  - have ALU instructions set flags
  - have branch instructions lookup flags

```
    cmp x4, x5    ; set flags
    b.ne #1000    ; branch if "not equal" flag is set
    mov r3, #0
...
1000: mov r3, #1
```

Handling functions

Software engineering mandates that long programs should be split in smaller functions.

We can support this at the hardware level by adding two instructions:

Handling functions

Software engineering mandates that long programs should be split in smaller functions.

We can support this at the hardware level by adding two instructions:

1. `call`: call a subroutine
   - Similar to a branch — on aarch64, 'branch and link' `bl` extends `b`
   - Saves the value of the *pc* into designated register
   - Sets the *pc* to the operand address

Handling functions

Software engineering mandates that long programs should be split in smaller functions.

We can support this at the hardware level by adding two instructions:

1. `call`: call a subroutine
   - Similar to a branch — on aarch64, 'branch and link' `bl` extends `b`
   - Saves the value of the *pc* into designated register
   - Sets the *pc* to the operand address

2. `ret`: return from a subroutine
   - Symmetric
   - Retrieves the old *pc* value from the designated register (saved during `call`)
   - Sets the *pc* to that old value

Handling function arguments

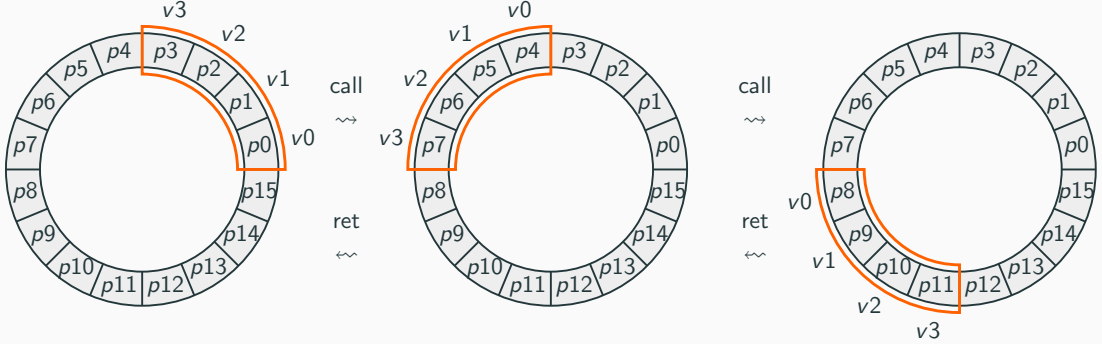
Different methods are used for **arguments**, depending on the hardware and the ABI:

- Store the arguments in **memory**
- Store the arguments in **special-purpose registers** in hardware
- Store the arguments in **general-purpose registers**

Many techniques also used to return result from function.

Handling function arguments: register windows

Some processors have a *register window* to **isolate** contexts and optimise function calls: programmer-visible registers v_i are virtual, and mapped onto different physical registers p_i according to an offset incremented by each function call:



Need to spill after a certain depth (here, 4).

An example Instruction Set

A **minimalistic** instruction set is the early RISC design known as *MIPS*.

It contains only 32 instructions
to address 32 registers
with 32-bit words.

Instruction	Meaning
Arithmetic	
add	integer addition
subtract	integer subtraction
add immediate	integer addition (register + constant)
add unsigned	unsigned integer addition
subtract unsigned	unsigned integer subtraction
add immediate unsigned	unsigned addition with a constant
move from coprocessor	access coprocessor register
multiply	integer multiplication
multiply unsigned	unsigned integer multiplication
divide	integer division
divide unsigned	unsigned integer division
move from Hi	access high-order register
move from Lo	access low-order register
Logical (Boolean)	
and	logical *and* (two registers)
or	logical *or* (two registers)
and immediate	*and* of register and constant
or immediate	*or* of register and constant
shift left logical	Shift register left N bits
shift right logical	Shift register right N bits

An example Instruction Set

The instruction set was designed to be **elegant**, without *frivolous* instructions.

It was also designed to be **easy to program/compile**, without arbitrary restrictions.

Principle of Orthogonality

Each instruction should perform a *unique* task without *duplicating* or *overlapping* the functionality of other instructions.

Instruction	Meaning
Data Transfer	
load word	load register from memory
store word	store register into memory
load upper immediate	place constant in upper sixteen bits of register
move from coproc. register	obtain a value from a coprocessor
Conditional Branch	
branch equal	branch if two registers equal
branch not equal	branch if two registers unequal
set on less than	compare two registers
set less than immediate	compare register and constant
set less than unsigned	compare unsigned registers
set less than immediate	compare unsigned register and constant
Unconditional Branch	
jump	go to target address
jump register	go to address in register
jump and link	procedure call

Remark: See book for additional floating-point instructions.

Implementation

Building a computer

Let's now switch our interest to building a computer capable of **executing** instructions:

- What are the major building blocks needed to create a processor?
- How are the building blocks arranged?
- What happens when an instruction is executed?

Building a computer

Let's now switch our interest to building a computer capable of **executing** instructions:

- What are the major building blocks needed to create a processor?
- How are the building blocks arranged?
- What happens when an instruction is executed?

We will build a **simplified** computer, focusing on how data moves through the **data path**:

- Processor with 32-bit **word size**
- Register file with 16 **registers**, storing 32 bits each
- Separate memories for instructions and data (**Harvard** architecture)
- **Byte-addressable** memory
- Hardware needed to execute for basic instructions: `load`, `store`, `add`, `jump`.

Remark: We will assume that program has been already **loaded** into memory.

Our instruction set

In our machine language, we specify instructions with an *operation* followed by *operands*.
Examples:

1. *Add* the contents of register 3 to register 2, store result in register 4.

```
add     r4, r2, r3
```

Our instruction set

In our machine language, we specify instructions with an *operation* followed by *operands*.
Examples:

1. *Add* the contents of register 3 to register 2, store result in register 4.

```
add    r4, r2, r3
```

2. *Load* register 1 with value from memory stored 20 bytes after the address in register 3

```
load   r1, 20(r3)
```

Our instruction set

In our machine language, we specify instructions with an *operation* followed by *operands*.
Examples:

1. *Add* the contents of register 3 to register 2, store result in register 4.

```
add    r4, r2, r3
```

2. *Load* register 1 with value from memory stored 20 bytes after the address in register 3

```
load   r1, 20(r3)
```

3. *Store* register 2 to memory address stored 12 bytes after the address in register 12

```
store  r2, 12(r12)
```

Our instruction set

In our machine language, we specify instructions with an *operation* followed by *operands*.
Examples:

1. *Add* the contents of register 3 to register 2, store result in register 4.

```
add    r4, r2, r3
```

2. *Load* register 1 with value from memory stored 20 bytes after the address in register 3

```
load   r1, 20(r3)
```

3. *Store* register 2 to memory address stored 12 bytes after the address in register 12

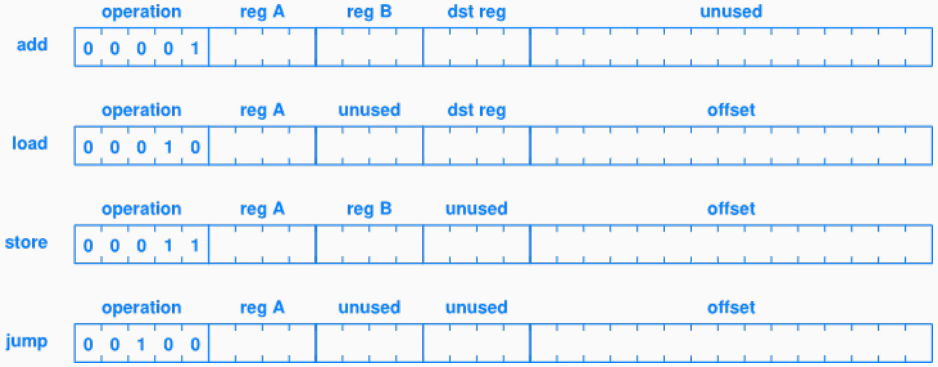
```
store  r2, 12(r12)
```

4. Branch to the address in memory with an offset of 60 to the contents of register 11

```
jump   60(r11)
```

Instructions in memory

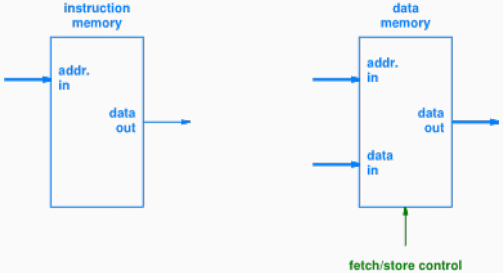
Now we can define a binary format to simplify the hardware:



Exercise: Assemble the instructions in the previous slide.

Instructions in memory

1. **Instruction** memory (read-only):
   - *Input*: 32-bit byte address (*program counter*)
   - *Output*: 32-bit data value (the four bytes starting at the specified address)
2. **Data** memory (RAM — can be read or written)
   - *Inputs*:
     - 32-bit byte address
     - 32-bit data (for writes)
     - 1-bit fetch/store signal (i.e. read/write)
   - *Output*: 32-bit data value (if the signal is fetch)



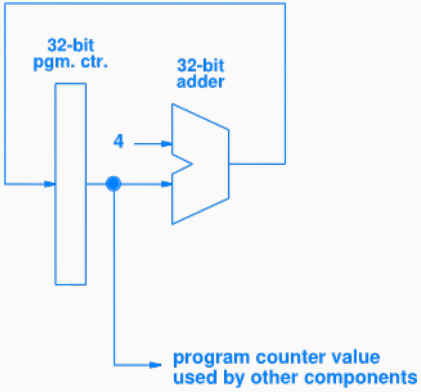
Data path design

Question: How to move to the next instruction?

Data path design

Question: How to move to the next instruction?

By adding 4 bytes to the *program counter* when clock signal says so.



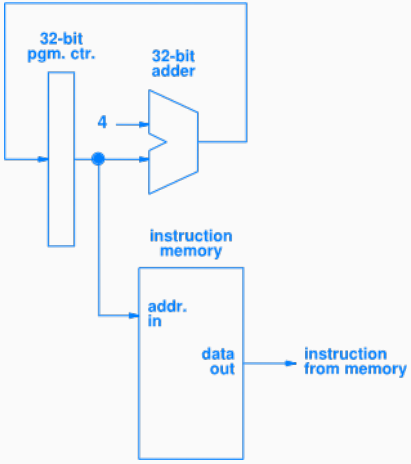
Data path design

Question: What should we connect to the output of the previous slide?

Data path design

Question: What should we connect to the output of the previous slide?

The **instruction memory**, taking a 32-bit address as input to produce a 32-bit instruction.



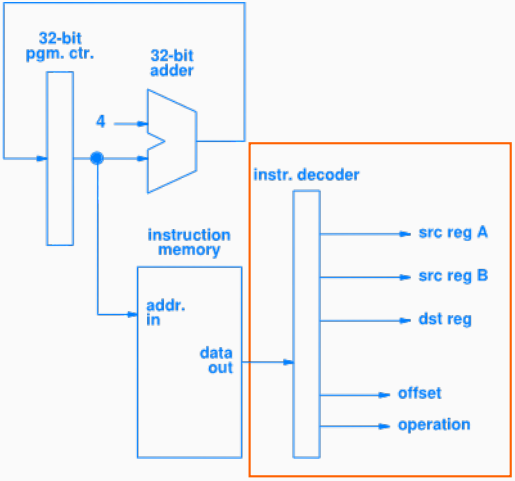
Data path design

Question: We have an instruction loaded from memory, now what?

Data path design

Question: We have an instruction loaded from memory, now what?

Break it in fields that are sent separate ways, depending on instruction format (*decoding*).



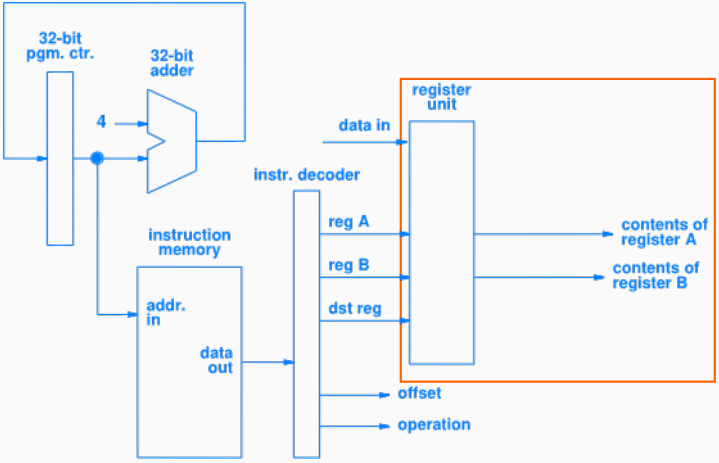
Data path design

Question: How to proceed, now that we have the operands decoded?

Data path design

Question: How to proceed, now that we have the operands decoded?

Plug-in the **register file**, with four inputs (registers+data) and two outputs for operands.



Control and coordination

A clock is used to synchronize all units, and additional control hardware coordinates overall data movement:

- Connects to each hardware unit
- Specifies when to transfer data

Control connections between controller and individual units are not shown because diagram illustrates data paths only.

Example: control lines (not shown) signal the register unit **when** to perform a fetch operation or a store operation.

Data path design

Question: What do we need to execute an arithmetic operation?

Data path design

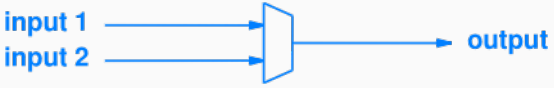
Question: What do we need to execute an arithmetic operation?
An Arithmetic Logic Unit (ALU) taking as inputs **either** two registers or register+offset.

Problem: How to support different types of operands?

Data path design

Question: What do we need to execute an arithmetic operation?
An Arithmetic Logic Unit (ALU) taking as inputs **either** two registers or register+offset.

Problem: How to support different types of operands?
Solution: Use a multiplexer to select between **possible** inputs based on *opcode*!



Data path design

Question: What are the possibilities for the ALU results?

Data path design

Question: What are the possibilities for the ALU results?
They can be either written to a register or used as a **data memory** address.

Two basic operations:

1. *Fetch* (read)
   - Place an address on the address input
   - Arrange for controller to signal fetch
   - Read a value from the data output
2. *Store* (write)
   - Place a value on the address input
   - Place a data value on data input
   - Arrange for controller to signal store

Confusing terminology... :-(

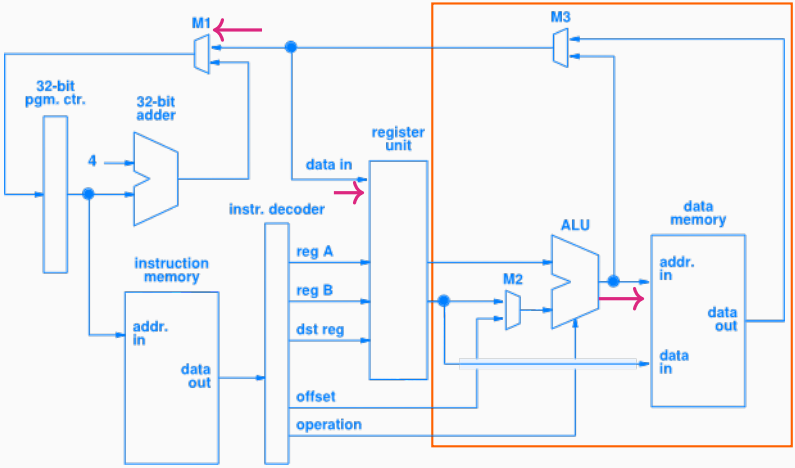
Data path design

Question: What are the possibilities for the ALU results?

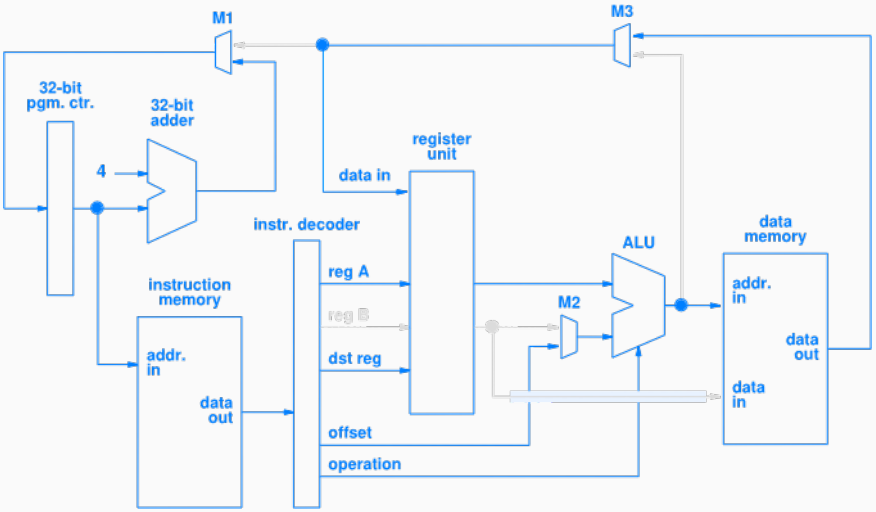
Data path design

Question: What are the possibilities for the ALU results?

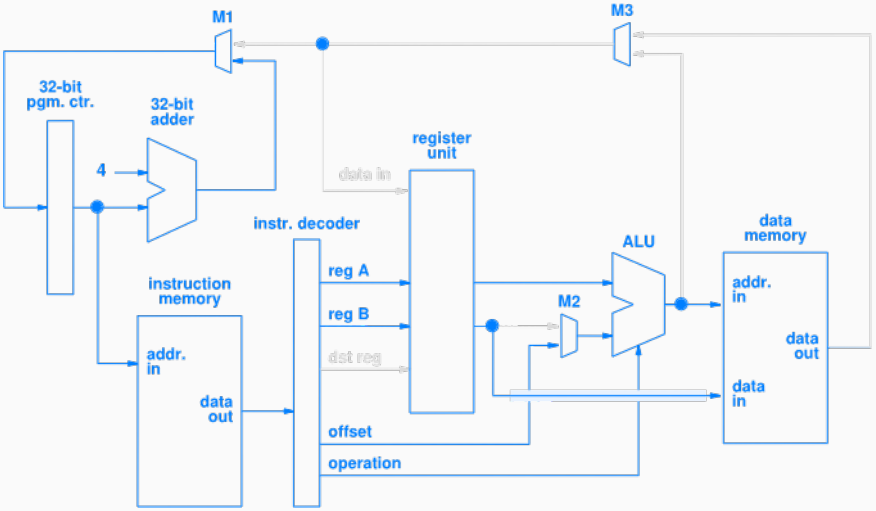
Written to a register, used as a data memory address, or as an instruction memory address.



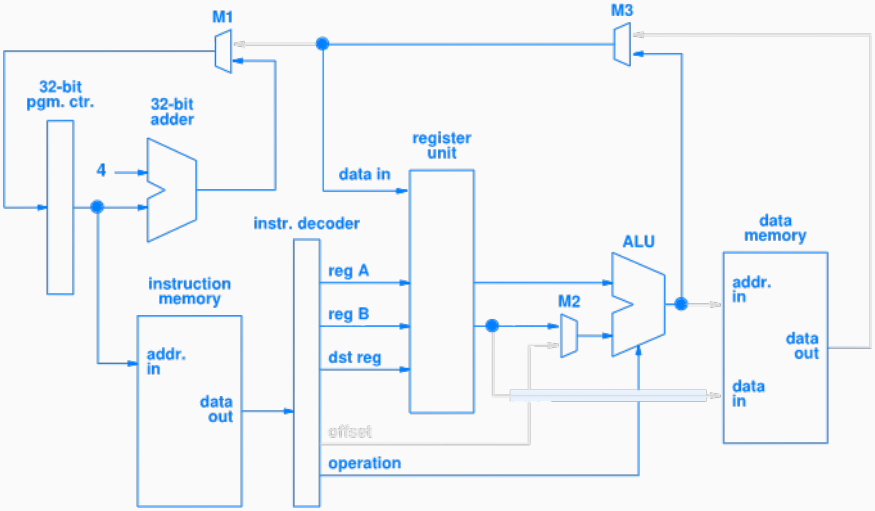
Data path executing a load



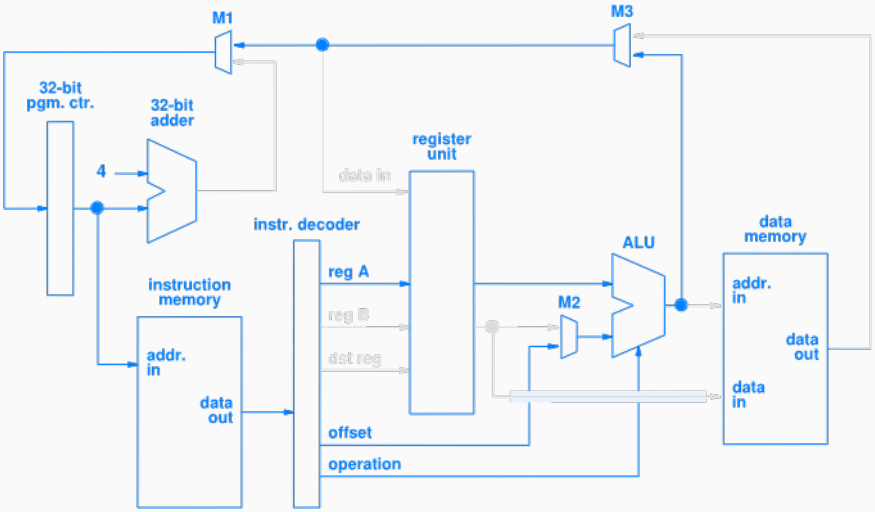
Data path executing a store



Data path executing an add



Data path executing a jump



Summary

The term **data path** describes interconnections among pieces of a processor.
Each data path contains multiple parallel wires.

Building **blocks** of a processor include:

- Program counter
- Decoder
- Register file
- Instruction and data memories
- ALU

A **multiplexer** passes one of its input data paths to the output data path.
Control signals determine which input a multiplexer selects at a given time
Using multiplexers, the processor chooses which data paths are **active** for a given instruction.