# Microprocessor Architectures

The object of this lecture is:

- to look at some different philosophies, architectures
- to examine a few other microprocessor ISAs
- to illustrate:
  - Not all processors are the same,
  - but there are usually many similarities.
  - CISC instruction set - less regular than RISC
    variable length op-codes (complicating state machine design)

The objective of this lecture is not:

- to learn another instruction set
- to understand all the details of *implementing* such a CPU

Assumes you ‘know’ some **Instruction Set Architectures** (ISAs) {ARM, Stump, MU0}

## Microprocessor architectures

Architecture (ISA) vs. microarchitecture (vs. Implementation)

Although the common usage may vary slightly, we will use “**architecture**” to describe the programmer-visible structure of a processor. This encompasses the registers which the programmer can read or write, the memory model and the instruction set, including its binary coding.

The architecture, once defined, needs to remain fixed to maintain compatibility with ‘legacy’ binaries. This does not, however, preclude the extension of the instruction set by adding new instructions – in unused ‘holes’ in the original coding – and this is common in any long-lived ISA.

The “**microarchitecture**” is the hardware view of the internal structure of the processor. This will often change significantly from generation to generation as the engineers strive for greater performance. The microarchitecture will contain registers unseen by the programmer: at the least there will be an instruction register (possibly many!) and, in all likelihood, numerous other registers holding temporary variables. (An example is the address register in the Stump microarchitecture pictured in the lab. manual.)

In cases such as the high-performance Intel Cores it is unlikely that the internal microarchitecture bears much resemblance to its distant ancestors when the ISA was defined. Here the instructions written by the compiler (or you) are translated on-the-fly by internal hardware into sets of simpler operations – and, possibly, reordered too – for issue in parallel to several execution engines. The external effect, however, is the same … except for the increase in speed.

The “**implementation**” may be the microarchitecture although it is often a combination of this with the physical details of a VLSI chip.

Example:

Intel have produced numerous upwardly-compatible microarchitectures of the ‘x86’ Pentium architecture, many of which have had publicly known names such as “Core”, “Nehalem”, “Sandy Bridge”, “Ivy Bridge”, …

Other Intel names have applied to implementations: thus “Clarkdale” is the “Nehalem” microarchitecture manufactured on a 32 nm process.

You don’t need to remember all these names!

## Instruction set evolution

Once upon a time each new microprocessor design came with its own, new instruction set. This made some sense because, as the number of transistors available had increased, there was lots more ‘processing power’ to employ.

The disadvantage to this is that it meant that any ‘legacy’ software at best had to be recompiled and at worst (e.g. if written in assembly language) rewritten. Even recompilation is impossible if you are a customer with only the object code available.

As computers became more widespread – and software more complex – it became increasingly desirable to retain ‘binary compatibility’ with previous processors. Adding new instructions was ‘allowed’ and new (or recompiled) software might exploit this but ‘backward compatibility’ requirements meant that it was useful to be able to run the old code as well.

Probably the most ‘extreme’ example of this is Intel’s x86 architecture which started as the 16-bit 8086 (1978) and has been supported by many generations as it has expanded into 32- and 64-bit processor implementations. Modern processors have many ‘modes’ of operation that facilitate backwards compatibility. They also provide a host of more recent instructions (and a simplified programming environment).

All of this is very painful for the microarchitect who has to support this legacy but it is a driving force in the architectures success, eclipsing other ISAs, even including Intel’s other attempts to provide something more ‘modern’.

Nowadays, completely new ISAs are fairly rare. Even though designing one is not that difficult (designing a ‘better’ one is a bit harder) the software legacy and the need for tool (compilers etc.) support makes it a hard to justify, economically.

**Reminder:**

**Source code**: what the programmer wrote, usually in a programming language (or more than one!) complete with variable names, comments, etc.

**Object code**: the ‘binary’ op-codes which the microprocessor interprets.

# Example ISAs

## Some definitions:

- ❑ Architecture (ISA): what the programmer can ‘see’
- ❑ Microarchitecture: the circuit-level (logical) design
- ❑ Implementation: the physical device

## Example ISAs – a selection

- ❑ ARM – 32-bit ‘RISC’ processor
  - ❍ *should be* familiar (to some extent!)
  - ❍ (although you haven’t been given the most recent developments)
- ❑ Intel 8080 – the direct precursor to x86:
  - ❍ ‘CISC’: many philosophical similarities
  - ❍ much simpler to present than IA-32 (x86)
- ❑ AVR – the small ‘RISC’ processor used in a family of microcontrollers, notably Arduino
  - ❍ 8-bit processor
  - ❍ Harvard architecture

## Some ISA-influencing factors

### Word size

Not always easy to determine (e.g. see later AVR example) the word size is usually taken as the width of the registers/ALU. A short word length takes less space (power) but is likely to need more operations to perform a ‘sensible’ size operation (e.g. 32-bit arithmetic).

There is also the issue of addressing. Most ‘8-bit’ processors use 16-bit addresses as 256 locations is very limiting. 32-bit processors have used 32-bit address spaces although this – rather than the need for 56-bit integers – is pushing new architectures to be ‘64-bit’.

### Memory bandwidth

With current technology processing is fast compared to memory access. This influences processor design in a couple of ways:

- ❑ Want numerous registers so that memory/load stores can be reduced: keep variables on-board the processor
- ❑ Want high instruction bandwidth. The majority of memory accesses are instruction fetches: if instructions are smaller then more can be fetched at once.

These require trade-offs: more registers imply more bits to specify them, thus longer instructions. Short instructions may reduce the functionality of each instruction, so *more* instructions may be needed.

To provide the ‘best’ instruction density – like any compression technique – means using short codes for frequent instructions and longer ones for infrequent operations. This, in turn, increases the complexity of instruction decode and issue and can impact speed or power consumption.

#### Superscalar processors

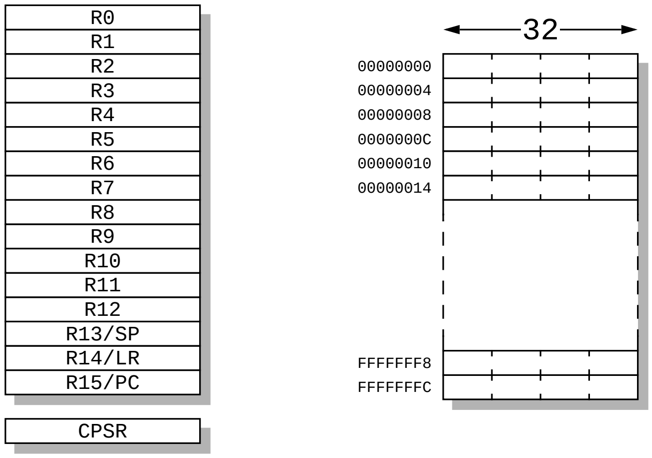
‘Superscalar’ processors are capable of executing more than one instruction at once. (Clearly, to do this requires more than one ALU, etc.) For this purpose it is necessary to identify not only the ‘next’ instruction but one or more after that. With fixed-length instructions this is easy! With variable length operations it is necessary to (at least partly) decode each instruction before the start of the subsequent one is found. This is slow and power-hungry.

## Some RISC ideas and ARM’s ‘compliance’

The basic RISC ideas are subject to some dispute! However here are some generally agreed principles and a view on how well ARM follows them; you might wish to add your own thoughts:

| Principle | ARM | Comments | Your thoughts? |
|---|---|---|---|
| Small instruction set | Around 10 basic instruction types | Could be simpler but reasonably straightforward | |
| Simple instruction decoding | See table of op-codes | Fairly regular thus quite easy | |
| Fixed length instructions | All 32-bit | Modern ARMs (‘Cortex’) use 16- and 32-bit instructions | |
| Large, regular register bank | 16x32-bit registers; a few with special functions | More registers than (e.g.) x86 but not as many as some RISCs | |
| Single cycle execution | For most operations but not (e.g.) LDM/STM | Pragmatic trade-off of usefulness vs. complexity? | |
| Memory access load/store only | Data processing instructions are separate from memory access. | Clearly follows this principle. | |
| Simple addressing modes | Register-indirect, indexed, pre-/post-increment … | Not really simple, no! | |
| Intended for pipelined execution | Yes … | … to some extent. | |
| Intended for fast clocking | Designed with shifter and ALU in series. | Originally (at least) relatively slowly clocked. Considerable effort to speed up. | |

# ARM ISA



- 32-bit datapath/ALU/operations
- Sixteen ‘general purpose’ registers
  - One acts as PC; one has extra function as ‘Link Register’
- Program Status Register holds flags (and other bits)
- 4 Gbyte memory address space – byte addressable but 32-bit words
  - I/O devices are memory mapped

## ARM

The slide shows a simplified ARM picture. ARM is based on RISC (Reduced Instruction Set Computer) ideas although it has some more complicated instructions too.

Most operations are single-cycle although ARM has some multi-cycle operations too: most notably the load- and store-multiple operations. (Any subset of the registers can be moved from/to memory with a single instruction although (of course) multiple memory cycles are needed.

Regardless of this, all ARM instructions are 32-bits long.

### Regular instruction coding

Below is the basic ARM op-code map: various later enhancements are omitted.

| | 31 | | | | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|
| #1 | cond | 000 | op | S | Rn | Rd | Shift | 0 | Rm |
| #2 | cond | 001 | op | S | Rn | Rd | Immediate | | |
| #3 | cond | 010 | op | L | Rn | Rd | Immediate | | |
| #4 | cond | 011 | op | L | Rn | Rd | Shift | 0 | Rm |
| #5 | cond | 100 | op | L | Rn | Register list | | | |
| #6 | cond | 101 | L | Offset | | | | | |
| #7 | cond | 110 | op | L | Rn | CRd | Copro | Immediate | |
| #8 | cond | 1110 | op1 | CRn | Rd/CRd | Copro | Op2 | CRm | |
| #9 | cond | 1111 | Number | | | | | | |

Example instructions according to category above:

```
ADD   Rd, Rn, Rm              ; #1 Data processing
SUBNE Rd, Rn, Rm, LSL #3      ; #1 Data processing
ANDS  Rd, Rn, #99             ; #2 Data processing
LDR   Rd, [Rn, #&40]          ; #3 Data transfer
STR   Rd, [Rn, Rm, LSL #2]    ; #4 Data transfer
LDMFD Rn, {R0-R5, PC}         ; #5 Multi-data transfer
B     label                   ; #6 Branch
BLEQ  method                  ; #6 Branch
SVC   33                      ; #9 OS call
```

#7, #8 are ‘coprocessor operations’ – e.g. used for floating point operations.

More ~~confusing~~ complete op-code maps for ARM are readily available. However this should serve to illustrate the coding principle and show some similarities with the Stump coding. (The Stump ISA is principally ~~derived from~~ inspired by ARM.)

### ‘True’ ARM architecture

The slide shows the *user* view of the ARM architecture. At *system* level it is somewhat more complicated because some of the registers are *bank-switched* according to the operating mode. Thus, for example, the operating system has a separate SP from the user.

To learn more about the ARM’s specific operation, come back in COMP22712!

### Instruction set evolution

Over time, more operations have been crammed into the ISA by exploiting ‘gaps’ in the instruction map. These are usually used to provide functions which are believed useful (especially by customers!) but are difficult (slow) to provide in software. Example:

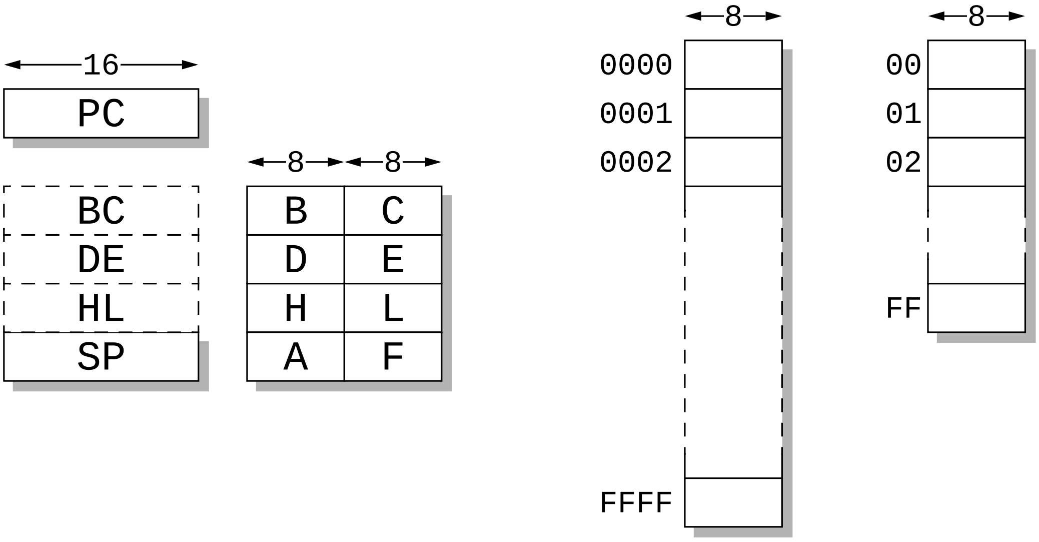
```
CLZ   Rd, Rm        ; Count Leading Zeros
```

returns (in Rd) the number of consecutive ‘0’ bits from the most significant end of Rm. (Try writing code to do this with the instructions you already know.)

Such instructions will not be generated by most compilers but can be provided in libraries.

However the biggest recent steps in ARM ISA evolution is the provision of a new instruction coding using a combination of 16- and 32-bit operations (‘Thumb’ and ‘Thumb2’ as supported by the ‘Cortex’ processors). This provides **higher instruction density** whilst (largely) retaining a compatible assembly language representation and using the same underlying microarchitectures. Many current products are able to decode both forms of instruction codes.

# Intel 8080



- ❑ 8-bit datapath/ALU/operations (some 16-bit extensions)
- ❑ Seven ‘general purpose’ 8-bit registers (plus flag ‘register’) – sometimes *paired*
  - ❍ but only one arithmetic ‘accumulator’
  - ❍ registers have ‘names’ not numbers … (but so what?)
- ❑ 64 Kbyte memory address space
- ❑ 256 byte I/O address space (extra, *alternative* address area)

## Intel 8080

It could be argued that the 8080 (1974) was the first ‘personal computer’ microprocessor. It served as the CPU in a number of microcomputer designs although its direct and indirect descendants later proved more popular.

The 8080 was used as the CPU for some early video games, including the extremely influential ‘Space Invaders’ machines.

The ‘direct’ successor was the Intel **8085** (1977) which used the same ISA (possibly with some extra, but undocumented, instructions) but represented a significant improvement in useability from the hardware point of view, requiring only one (rather than three) power supply voltages and fewer support chips.

A more successful processor, the **Z80** (1976) was produced by Zilog. The Z80 was **binary compatible** with the 8080 – i.e. it could run the same object code – but provided a host of additional instructions as well as extra registers. Furthermore, it included a number of features (such as DRAM refresh, and if that means nothing to you, don’t worry about it) which made building a computer around it easier.

The Z80 was widely used for many years including in a range of home computers from the early ‘80s onwards. In the UK the most popular were the **Sinclair** ZX80, ZX81 and, particularly, the **Spectrum**. Shortly afterwards they were used for a range of **Amstrad** machines as well as numerous other, less well-known systems.

Z80s were manufactured under licence by several companies in the USA, Japan and Korea; unlicensed clones (this was still the Cold War era) were produced in East Germany, Romania and the USSR.

Later developments by several companies improved the silicon integration of the parts as well as, at times, adding extra instructions. There were also some similar but non-compatible developments.

## Intel 8086

Intel’s next offerings in this market were the **8086** and **8088**. These devices shared an architecture and instruction set but had different external interfaces in that the 8086 had a 16-bit data but whereas the 8088 had only 8 data pins. Their binary codes are completely different from the 8080 but they were marketed as ‘assembly language compatible’. In practice it made more sense to recode most programmes but the legacy in the internal architecture is readily apparent.

The 8088 (originally) was adopted by IBM for its contribution to the ‘PC’ market. The rest, as they say, is history!

Of course the 8086 gave rise to a succession of chips which, whilst retaining binary compatibility have continued to add instructions, registers etc. These adopted later names rather than numbers (for legal/marketing reasons) and have turned into the IA-32 series of devices.

A ‘popular’ ‘operating system’ written for the 8080 was CP/M. CP/M was later ported to other processors such as the 8086 where it appears to have inspired a rival product called DOS. The acceptance of DOS for the first IBM PCs proved a great success for a (then small) software company called ‘Microsoft’.

# 8080 Instruction Set

- Registers quite numerous (for the era)
- Accumulator-based operation
  - Most arithmetic/logical results end up in a dedicated register (‘A’)
  - Therefore most registers are operand stores
- Limited memory addressing (compared to some near-contemporaries)
  - some direct addressing
  - indirect addressing with HL – and sometimes other register pairs
  - capable of (e.g.) incrementing a memory location (1 instruction)
- ‘Typical’ set of arithmetic/logical/shift operations {ADD, ADC, SUB, SBC, AND, XOR, OR, CP}
  - No multiplication (or division) instructions: too expensive in this era
- Control by conditional branch (call, return) on status flag state
- Regular coding in places, other instructions a bit more ‘shoehorned’
- Different choice of assembler mnemonics, e.g.
  - ‘LD’ is used for all ‘move’s: {mem⇒reg, reg⇒mem, reg⇒reg}
  - ‘CP’ instead of ‘CMP’ for compare

## 8080 op-codes

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | nop | ld bc, WW | ld (bc), a | inc bc | inc b | dec b | ld b, BB | rlca | — | add hl,bc | ld a,(bc) | dec bc | inc c | dec c | ld c,BB | rrca |
| 1 | — | ld de, WW | ld (de), a | inc de | inc d | dec d | ld d, BB | rla | — | add hl, de | ld a, (de) | dec de | inc e | dec e | ld e, BB | rra |
| 2 | — | ld hl, WW | ld (WW), hl | inc hl | inc h | dec h | ld h, BB | daa | — | add hl,hl | ld hl, (WW) | dec hl | inc l | dec l | ld l, BB | cpl |
| 3 | — | ld sp, WW | ld (WW), a | inc sp | inc (hl) | dec (hl) | ld (hl), BB | scf | — | add hl,sp | ld a, (WW) | dec sp | inc a | dec a | ld a, BB | ccf |
| 4 | ld b, b | ld b,c | ld b, d | ld b, e | ld b, h | ld b, l | ld b, (hl) | ld b, a | ld c, b | ld c, c | ld c, d | ld c, e | ld c, h | ld c, l | ld c, (hl) | ld c, a |
| 5 | ld d, b | ld d, c | ld d, d | ld d, e | ld d, h | ld d, l | ld d, (hl) | ld d, a | ld e, b | ld e, c | ld e, d | ld e, e | ld e, h | ld e, l | ld e, (hl) | ld e, a |
| 6 | ld h, b | ld h, c | ld h, d | ld h, e | ld h, h | ld h, l | ld h, (hl) | ld h, a | ld l, b | ld l, c | ld l, d | ld l, e | ld l, h | ld l, l | ld l, (hl) | ld l, a |
| 7 | ld (hl), b | ld (hl), c | ld (hl), d | ld (hl), e | ld (hl), h | ld (hl), l | hlt | ld (hl), a | ld a, b | ld a, c | ld a, d | ld a, e | ld a, h | ld a, l | ld a, (hl) | ld a, a |
| 8 | add a, b | add a, c | add a, d | add a, e | add a, h | add a, l | add a, (hl) | add a, a | adc a, b | adc a, c | adc a, d | adc a, e | adc a, h | adc a, l | adc a, (hl) | adc a, a |
| 9 | sub a, b | sub a, c | sub a, d | sub a, e | sub a, h | sub a, l | sub a, (hl) | sub a, a | sbc a, b | sbc a, c | sbc a, d | sbc a, e | sbc a, h | sbc a, l | sbc a, (hl) | sbc a, a |
| A | and a, b | and a, c | and a, d | and a, e | and a, h | and a, l | and a, (hl) | and a, a | xor a, b | xor a, c | xor a, d | xor a, e | xor a, h | xor a, l | xor a, (hl) | xor a, a |
| B | or a, b | or a, c | or a, d | or a, e | or a, h | or a, l | or a, (hl) | or a, a | cp a, b | cp a, c | cp a, d | cp a, e | cp a, h | cp a, l | cp a, (hl) | cp a, a |
| C | ret nz | pop bc | jp nz, AAAA | jp AAAA | call nz, AAAA | push bc | add a,BB | rst 00 | ret z | ret | jp z, AAAA | — | call z, AAAA | call, AAAA | adc a,BB | rst 08 |
| D | ret nc | pop de | jp nc, AAAA | out (BB),a | call nc, AAAA | push de | sub a,BB | rst 10 | ret c | — | jp c, AAAA | in (BB),a | call c, AAAA | — | sbc a,BB | rst 18 |
| E | ret po | pop hl | jp po, AAAA | ex (sp), hl | call po, AAAA | push hl | and a,BB | rst 20 | ret pe | jp (hl) | jp pe, AAAA | ex de, hl | call pe, AAAA | — | xor a,BB | rst 28 |
| F | ret p | pop af | jp p, AAAA | di | call p, AAAA | push af | or a,BB | rst 30 | ret m | ld sp, hl | jp m, AAAA | ei | call m, AAAA | — | cp a,BB | rst 38 |

Some instruction coding is quite straightforward:

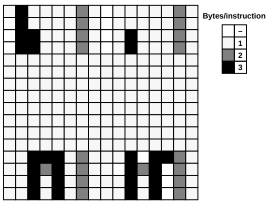
```
                 7             0
LD   Rd, Rs    | 01 | ddd | sss |
<op> A, Rs     | 10 | op  | sss |
```

```
Register codes
| 000 |B   | 100 |H
| 001 |C   | 101 |L
| 010 |D   | 110 |(HL)
| 011 |E   | 111 |A
```

Other areas are less regular.

The ‘register’ code (HL) is a memory indirect mode; the register pair forms a 16-bit address from where the operand is fetched. It can be used ‘like’ a register in that the fetched value can be used as an operand in an arithmetic operation by the same instruction (e.g ‘`ADD A, (HL)`’). It can even be used as a destination – e.g. ‘`INC (HL)`’ will add 1 to the memory location without altering registers. Of course this sort of instruction still requires the read and write bus operations so it takes longer than an internal register operation. This form of instruction is distinctly ‘CISC’.

Instructions vary from 1-3 bytes in length. The figure on the right depicts the number of bytes/instruction in the same plan as the table above; *some* regularity is apparent.



The 2-byte are mostly immediate operations, either loading an 8-bit value (the second byte) to a register or using the value in an arithmetic/logical operation on A. The exceptions are ‘IN’ and ‘OUT’ where the extension byte is an 8-bit I/O space address.

The 3-byte operations are mostly jump/call instructions where the 16-bits of operand form a target address; there are also 16-bit (register pair) loads, both immediate and absolute (direct).

Some op-codes were not used.

# Variable Length Instructions

- 8080 instructions are strings of 1-3 bytes
  - The first byte is the ‘real’ op-code and specifies **what** is done with which **register(s)**
  - Subsequent bytes act as operands e.g.

| | |
|---|---|
| — an 8-bit immediate (literal) operand | `ADD A, 0x23` |
| — an 16-bit immediate (literal) operand | `LD HL,0x1234` |
| — an 16-bit absolute address | `JP label` |
| — an 8-bit I/O port address | `IN A, 0x80` |

  - The first byte has enough information to decode the whole instruction behaviour
  - Later (Z80) extensions introduced 4-byte codes as well

Some ISAs are more complicated

- x86 instructions are also byte strings varying from 1 to 17 (or longer?) bytes
  - Not all the information is contained in the first byte
  - Decoding is … complicated

‘ret’ is a method return, like “MOV PC, LR” on ARM

It is needed because the PC is 16 bits wide and not generally accessible.

## 8080: more details

[N.B. the **concepts** are important here: DON’T memorise the details.]

The 8080 typifies an earlier generation of processor architectures but also provides a relatively simple view of the ISAs of descendant CISCs (Complex Instruction Set Computers); the most prominent contemporary family are the IA-32 (x86) devices.

- Complex operations such as subroutine (method) ‘`call`’
- Variable length op-codes
  - from 1 to 3 bytes for 8080
- Extensions allow arbitrary 8-/16-bit values to be used as appropriate
  - cf. ARM has a limited selection of literal values available
- Variable execution times
  - 4-17 clock cycles for original 8080
  - 1-5 (8-bit) memory transfers involved – likely to be the limiting factor
- Can provide high code-density
  - provided that frequent instructions have short codes

The specificity of many of the operations (e.g. destination ‘A’ for most arithmetic/logical operations allows shorter op-codes but less flexibility.

Sometimes there are multiple ways of expressing an operation:

- `ADD   A,1` ; general add of literal: 2 bytes
- `INC   A` ; special operation: 1 byte

or

- `LD    A,0` ; uses literal: 2 bytes
- `XOR   A,A` ; programmer’s ‘trick’: 1 byte (think about it)

**Differences from (e.g.) ARM**

8080 jump instructions (‘jp’) are all to absolute addresses: i.e. the complete 16-bit destination address follows the initial 8-bit op-code (LSB first). ARM Branch (‘b’) instructions are all PC-relative; they have a 24-bit (word) offset which allows branches ±8 Mwords – less than the whole address space.

Statistically, most branches are short-range (implementing if … else … or loops) so coding them with fewer bits makes ‘economic’ sense. The Z80 extensions to 8080 added ‘relative jumps’ with an 8-bit signed offset – thus taking two bytes instead of three – which shortened (and speeded up) code. (IA-32 supports both and has relative jumps with both short and long offsets.)

Relative branches are also **relocatable** – i.e. the code can be moved to a different address and still run. With absolute branches, the first jump in a moved section of code would leap back to its original target. Relocatability of code is sometimes useful, especially if the system has no memory management.

The ‘**`call`**’ instruction on 8080 (& IA-32 etc.) is like the ARM ‘`bl`’ but carries out significantly more operations.

- The (absolute) target address is fetched and held internally
- The PC – now pointing at the subsequent instruction – is pushed onto the stack
  - This involves two successive 8-bit store operations decrementing SP before each
- The target is loaded into the PC

There is an accompanying subroutine return (‘`ret`’) instruction which pops the PC (requiring two cycles with SP post-increment).

Conditional operations are confined to branches (including `call` and `ret`). The conditions are less sophisticated than ARM, testing the state of a single flag.

| | | |
|---|---|---|
| ❑ nz/z | non-zero/zero | Zero flag - equality |
| ❑ nc/c | no carry/carry | Carry flag |
| ❑ po/pe | parity odd/ parity even | Parity - no ARM equivalent |
| ❑ p/m | plus/minus | Sign (Negative) flag |

There are some (limited) 16-bit operations using (e.g. ‘add hl, bc’). In the initial implementations these were performed with two 8-bit addition cycles.

### Register indirect addressing mode

One of the 8080 ‘registers’ (as coded in the instruction set) is a really a register indirect addressing mode. The 16-bit address is taken from a pairing of 8-bit registers. This allows instructions such as: `add a, (hl)`.

`(hl)` is the only ‘general’ indirect mode. There are also `(bc)` and `(de)` but these are only useable for loading or storing ‘`a`’, not for other operations.

# Sequencing

The exact sequencing of operations depends on the microarchitecture.

- The following examples are used to *illustrate some possible* state sequences without trying to optimise too much.
- The original microarchitecture took significantly more cycles than are indicated here.
    - Technology limits meant numerous cycles needed for ‘internal’ operations.
    - Cheap transistors/wiring help with speed-up

Sequencing is done by a Finite State Machine which uses the IR to determine its path.

The FSM can become quite complex

- sometimes complexity may be traded for performance
- more often the other way around though!

## Sequencing

In the following description some more registers are used which are non-architectural (i.e. not ‘visible’ to a programmer).

- IR the instruction register - holds (the first byte of) the op-code.
- W, Z working registers which may act as a 16-bit pair.

```
0000 78         ld  a, b        ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; ALU:     A  := B;

0001 04         inc b           ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; ALU:     B  := B + 1;

0002 C6 12      add a, 12       ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; OP1:     Z  := [PC]; PC := PC + 1;
                                ; ALU:     A  := A + Z;

0004 32 00 10   ld (1000), a    ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; OP1:     Z  := [PC]; PC := PC + 1;
                                ; OP2:     W  := [PC]; PC := PC + 1;
                                ; MEM0:    [WZ] := A;

0006 BB         cp  a, e        ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; ALU:     flags set by (A - E);

0007 CA 68 24   jp  z, 2468     ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; OP1:     Z  := [PC]; PC := PC + 1;
                                ; OP2:     W  := [PC]; PC := PC + 1;
                                ; JUMP:    IF (Zflag == 1) PC := WZ;

000A 96         sub a, (hl)     ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; MEM0:    Z  := [HL];
                                ; ALU:     A  := A - Z;

000B C5         push bc         ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; ALU:     SP := SP - 1;
                                ; MEM0:    [SP] := B;
                                ; ALU2:    SP := SP - 1;
                                ; MEM2:    [SP] := C;

000C CD 57 13   call 1357       ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; OP1:     Z  := [PC]; PC := PC + 1;
                                ; OP2:     W  := [PC]; PC := PC + 1;
                                ; ALU:     SP := SP - 1;
                                ; MEM1:    [SP] := PC(high byte);
                                ; ALU2:    SP := SP - 1;
                                ; MEM2:    [SP] := PC(low byte);
                                ; JUMP:    PC := WZ;

000F C9         ret             ; FETCH:   IR := [PC]; PC := PC + 1;
                                ; MEM0:    Z := [SP];
                                ; ALU:     SP := SP + 1;
                                ; MEM1:    W := [SP];
                                ; ALU2:    SP := SP + 1;
                                ; JUMP:    PC := WZ;
```

## State machine code

The Verilog code to implement the machine here (and following) is not specified exactly here. A next-state *decision* is made in the `fetch` state, before the IR is loaded.

To make this ‘clean’ code would involve adding an extra `decode` state, after the fetch.

```
always @ (posedge clk)
  case (state)
    ‘FETCH:  begin
             IR <= mem_data_in     // Could be done here ...
             PC <= PC + 1;         // ... or in separate ‘always’ block
             state <= ‘DECODE;
             end

    ‘DECODE: begin
             case (IR[7:6])
               2’b00: ...
               2’b01: ...
               2’b10: if (IR[2:0] == 3’b110) state <= ‘MEM0;   // (HL)
                      else                   state <= ‘EXEC0;  // reg.
               2’b11: ...
               end

    ...
```

You can, no doubt, think of other approaches which may save the ‘extra’ state. For example:

- Do the first execution step in a ‘`case`’ within the ‘`‘decode`’ state.

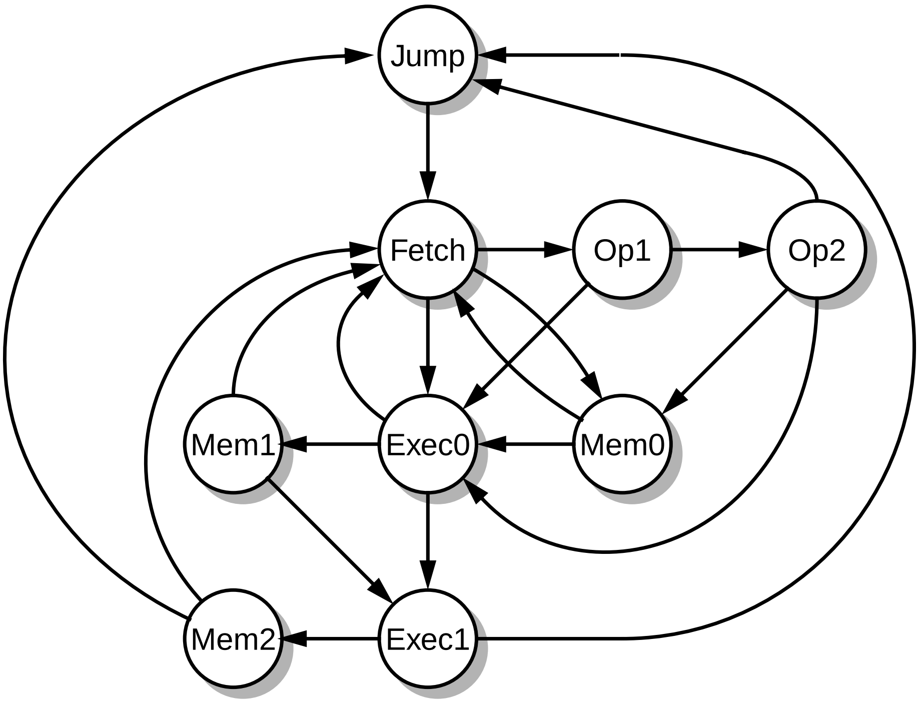
or

- Make a state decision in ‘`‘FETCH`’ on the IR *before* it is latched i.e. use ‘`mem_data_in`’ in this step only (IR thereafter).

or

- …

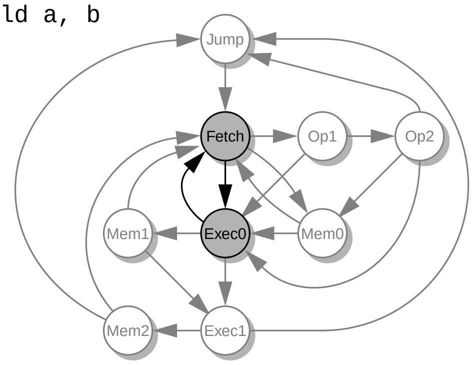
# Possible state diagram



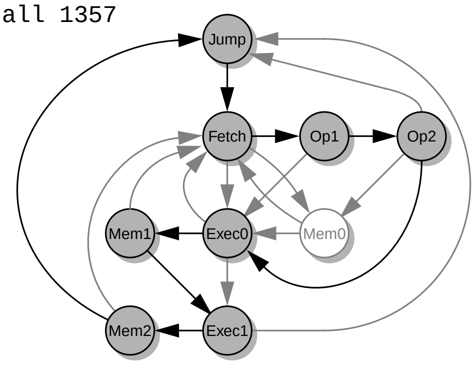
- ❑ The real point here is that the behaviour is quite complicated!

## Possible execution paths

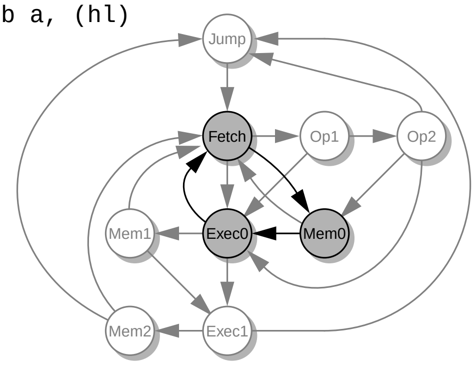
`ld a, b`



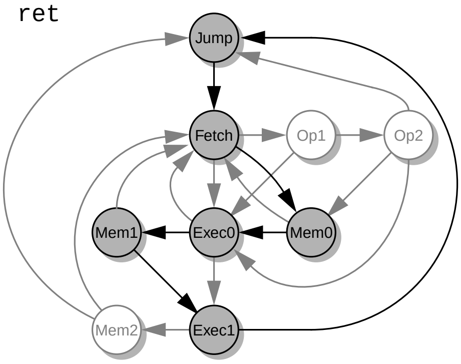
`call 1357`



`sub a, (hl)`



`ret`



# Optimisation

This is the area where the microarchitect has the most input. Ideally:

- Use every cycle for something useful
- Get in as much parallelism as is practical
- Try to divide up the system (as sensible) to allow higher clock speeds

The chief trade-off is probably resources vs. speed: more resources ⇒ higher speed (hopefully?)

**Example:**

- The Stump in the lab. achieves less than 0.5 IPC
  - 2 cycles for most instructions, 3 for LD/ST
- The execute cycle never touches memory — this is wasteful
- Fetch the next instruction *concurrently* with the current execution
  - Nearly doubles performance
  - Needs separate PC incrementer – and probably PC separating from register bank
  - Still need memory cycle for LD/ST
  - Need to wait for target calculation when branching

## Optimisation

The implementor's challenge is usually to execute instructions in the fewest cycles with the resources available. In early microprocessors resources were scarce – hence, for example, the 8-bit data path. Nowadays logic is cheap so (usually) more resources can be spent.

Parallelism is the key to accelerating execution: if more things can be done in each cycle then fewer cycles are required to complete the instruction. We have assumed some parallelism already in the instruction fetch: the memory read and the PC increment have been done in parallel.

> Also note that the PC increment is a 16-bit operation; with an 8-bit ALU it would be necessary to increment one byte of this and (every 256 times) insert another cycle to increment the high byte to propagate the carry. The option to skip the last cycle when not needed (when adding a '0' carry – most of the time) speeds up the fetch significantly, on average.

There are multiple opportunities for parallelism in the execution schemes described above. When identifying these it is important also to note **dependencies**; e.g. it is not possible to add a value from memory *before* it is fetched.

Typically the biggest limitation on a processor are its memory accesses. If the 8080 has a single 8-bit read/write port to memory then the fastest it can go is one cycle/byte read or written.

This suggests simple operation, such as `add a, b` could be completed in a single cycle because it is a one byte instruction. However the instruction must be fetched (from memory) before it can be executed. To achieve single-cycle operation the *next* instruction must be fetched in parallel with the current execution, so it is ready in the next cycle. This is known as '**prefetch**' and is the first step in **pipelining** instruction execution.

Note that this has already increased the demand for resources. If the 'add' executes in parallel with a fetch then both an adder and an incrementer (to increment the PC) are needed at the same time; one ALU cannot now do both jobs.

> The slowest 8080 instruction is still going to be 'call' because it needs five memory cycles: three to fetch the instruction and two to push the halves of the old PC.

### Pipelining

The concept of pipelining should already be familiar: place registers at intervals along the processing path so that other operations can be started before the first one finishes. Pipelines can speed up **throughput** but will also increase **latency**, especially if they are not well **balanced** (i.e. each stage takes close to the same amount of time to evaluate).

Pipelines are effective in processors but can suffer from stalls due to **dependencies** – such as the need to wait for a result from one instruction before starting the next. To alleviate this, sometimes it is possible to issue instructions **out-of-order**: this comes with more complications though!

Pipelines also suffer when branches are taken – because the instructions ingested may be from the wrong place and need flushing out – unless there is an effective **branch predictor**.

### Wider memory

One way of overcoming the 'memory barrier' is to widen the memory. A 16-bit bus allows twice as many bytes/s to be moved. This technique is widely used now that memory is affordable. When retro-fitted to an old architecture there are certain other problems: a 16-bit word could contain one, two, one-and-a-bit or even less than one instruction, depending on the program. The complication of the implementation just increased again!

This is not an abstract issue. IA-32 (for example) does this extensively.

**Performance measurement**

Although it is significant the clock frequency (speed) is not the only factor affecting performance. Microarchitects will strive to increase the IPC (Instructions Per Clock) figure for a particular design (or, conversely, decrease the CPI).

For a scalar (one ALU) processor the target is 1 IPC but, due to other issues, this is unreachable. Figures such as 0.95 IPC (it's an *average* over a set of benchmarks) should be achievable in pipelined processors with separate buses.

Superscalar processors can achieve >1 IPC

# Z80 extensions

The Z80 was an 8080 upwardly compatible processor. It added operations such as:

- PC-relative conditional jumps (8-bit signed offset)
- More shift and rotate operations
- Bit operations
- Extra registers - some used for indexing with 8-bit literals
- Additional interrupt modes
- [Easier-to-use hardware, too]

The ISA extensions were achieved by:

- New instructions filling some unused ‘holes’ in the op-code map
- Some unused codes acting as ‘escape’ codes to extra 8-bit maps
  - CB ⇒ shifts and bit operations
  - ED ⇒ more ‘16-bit’ arithmetic/loads & other miscellany
  - DD/FD ⇒ alter ‘HL’ instructions to use extra index registers

The Z80 was very popular as a CPU in home computers in the 1980s - e.g. Sinclair Spectrum

## Extending an ISA

### 8080

The problem with extending an instruction set after its initial release is difficult. The **Z80** used ‘holes’ in the instruction map. As there were not many of these some were employed as ‘escape codes’ – a bit like a shift key – which, if encountered indicated another byte was to be fetched and this provided another set of 256 codes.

This technique has been employed extensively in the development of the IA-32 instructions – which sometimes has multiple ‘escapes’ and prefixes.

Another technique quite commonly employed is the setting of some processor state which changes its behaviour. On the Z80 this was used, but only to alter the interrupt behaviour. It is employed extensively in x86 to provide completely different new instruction sets.

### ARM

ARM has also been extended although, in most cases, this has disrupted the ‘cleanliness’ of the instruction coding. The sort of thing which can be done:

Exploit bits which were previously ignored.

```
    MOV   R0, R2      ; is coded in a similar manner to
    ADD   R0, R1, R2
```

but one operand field was initially ignored. This can be *defined* to be (e.g.) zero and if it isn’t the meaning of the whole instruction may be completely different.

(The ‘destination’ field of comparison instructions gives another opportunity.)

With ARM all instructions have a 4-bit condition code. There are eight conditions and their complements. One of the eight is ‘always’ so there is a complementary ‘never’. This makes $\frac{1}{16}$ of the instruction space (268435456 potential instructions) into NOPs. Later it was decided to make these cases into ‘always’ instructions and use some of this space more constructively.

## Coprocessors

ARM (and other processors) have also been extended by the addition of ‘coprocessors’ Originally a coprocessor was a separate chip which ‘piggybacked’ the CPU bus and (when present) could execute instructions not implemented on the CPU. A common application was(/is) floating point operation where a coprocessor will have its own registers and ALU(s).

Coprocessors are now implemented on-chip but are still not uncommon. In x86 the floating point unit has now become a fixture. ARM – which targets a different market in mobile devices – has a floating point unit but it is not always included on the silicon for cost/power reasons.

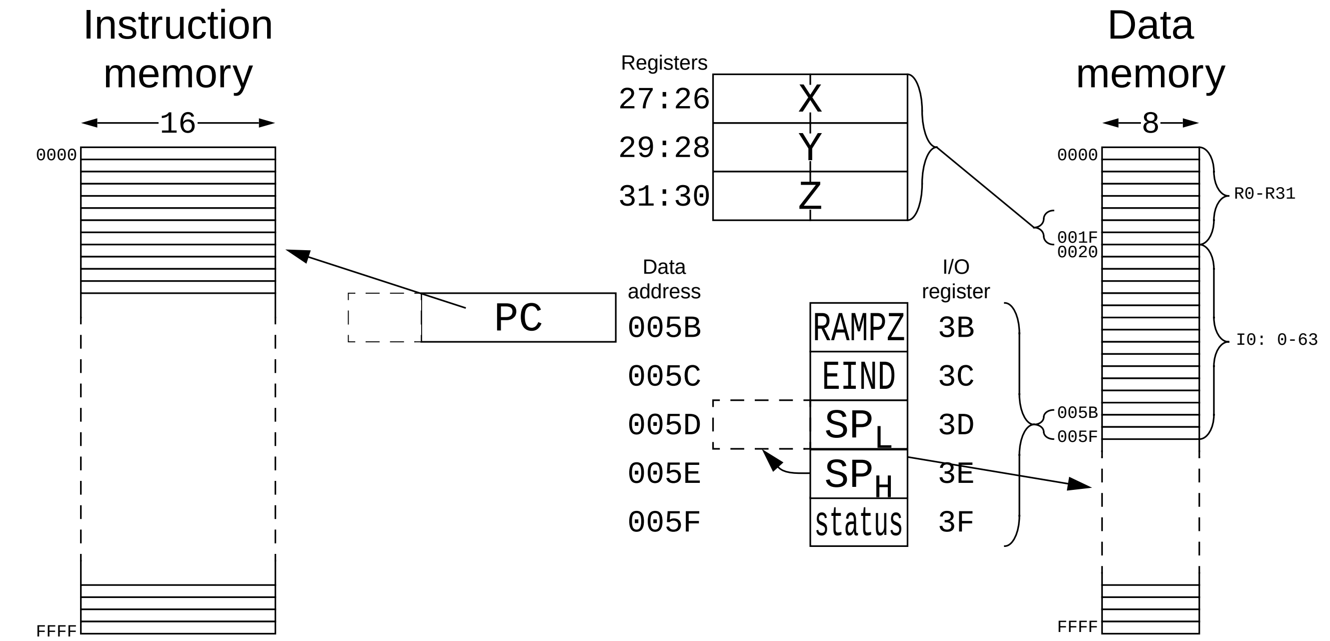
Coprocessors can be used to extend functionality in numerous ways. Another common extension is to provide a SIMD multimedia unit such as MMX/SSE in x86 or NEON in ARM. (You can use this as a starting point if you want to investigate this further; the scope goes beyond this module.)

# AVR

Harvard: 64K (or 4M) of 16-bit instruction address space plus 64K of 8-bit data address space

(Almost fixed length instructions)

No ‘general purpose’ registers … but …



Subsequently extended to 22 bits

## AVR

A popular ‘modern’ (1996) microcontroller processor, the AVR is aimed more at the cheap/simple market than being a workstation processor. Devices are small, cheap and – by contemporary standards – slow, with maximum clock speeds in the ‘tens of MHz’ range. (This, of course, still offers much more processing power than the original 8080 etc.)

AVRs are used as the CPU for the popular **Arduino** single-board computers.

ARM, 8080, IA-32, (Stump, MU0) are all **von Neumann** architecture: they have a single (main) memory space which contains **both code and data**. The application of any particular byte or word depends on its interpretation.

AVR is an example of a **Harvard** architecture where there are **separate address spaces for code and data**. There are several advantages to this:

- ❏ Instructions do not have to be the same length as the data; it is sometimes convenient for them even to be not power-of-two in size.
- ❏ Because they have separate buses, instructions and data can be transferred simultaneously. In a von Neumann architecture instruction fetching might have to pause whilst data transfers complete.

On the other hand there are also disadvantages:

- ❏ Code and data cannot (easily) be interchanged. This tends to rule out techniques such as run-time compilation.

In one sense the AVR does not have ‘general purpose’ registers: it has 32 8-bit ‘registers’ which are used in operations but these are *aliases* for locations in the memory map. Thus, ‘R3’ is also address ‘0003’. Some of these registers have special functions: this includes the action of register pairs to form 16-bit pointers (as pairs aliased as ‘X’, Y, & ‘Z’).

A register and a memory location being aliased does not compel the implementation to keep everything in memory: probably, in practice, references to certain memory locations are ‘trapped’ and rerouted to internal registers.

The architecture has a similar set of 64 ‘I/O registers’, also aliased to the address space. Despite the name, some of these fulfil ‘normal’ processor functions such as a program status register and a pair which, together, act as a stack pointer. These are numbered 0-63 but alias to locations $0020_{16}$-$005F_{16}$.

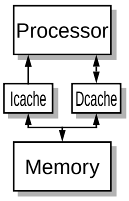
The only ‘register’ separate from the memory is the PC; initially 16-bit this has been extended in later implementations to allow more code memory.

In one sense the AVR’s register numbers are simple short-form addresses; they supply 5 bits and the other 11 are all ‘0’. The idea of using ‘short’ addresses to get at particular memory locations is not new. Other processors have employed similar techniques to allow shorter (faster, if fetching variable-length instructions) addresses for a limited set of locations. Sometimes the upper bits are constant (as here); on occasion they have been defined (e.g. by a writeable register) so the ‘fast’ page can be moved.

Flags (include) N, Z, C, V and some less usual ones such as a half-carry (used for BCD operations – 8080 also has this) and a ‘sign flag’ (N xor V) which is an AVR peculiarity.

The AVR32 processors are designed for a similar market but have a different ISA.

### Harvard microarchitecture



Although a processor like ARM has a von Neumann architecture to the user it is common for high-performance implementations to have separate instruction and data buses to give the benefits of parallel code and data fetches. This is supported by separate caches, at least at ‘level 1’; the memory model is unified before the ‘main memory’ is reached.

A consequence of this which *is* visible in software is the need for ‘memory barriers’. If the processor writes some code (in the data cache) it must ensure this is also forced out to where thew model is unified before trying to execute it.

# AVR Instruction Set

- Most op-codes are 16-bit
  - Exceptions: operations with long immediate fields, (JMP, CALL, direct addressing)
- Load/store architecture
  - Primarily using register (pair) indirect addressing
  - Additional instructions for access to program memory
  - Some ‘register’/‘I/O’ transfers
- 8-bit integer arithmetic/logical/shift operations
  - Includes an 8x8 multiplier
  - Some (limited) 16-bit support
- Absolute and PC-relative jumps and calls: also an ‘indirect’ jump
- Decision making using:
  - conditional (relative) branches on status bits (set/clear)
  - test-and-skip on I/O bits

## AVR instruction set

The AVR instruction set is RISC-like, based on load/store memory access and operating on registers. Most instructions are 16-bits long. That said it contains a few 2-word (32-bit) instructions, used for (e.g.) `JMP` which has a 16 (or 22) bit absolute address and can thus jump to any location in instruction memory. (There is a shorter, `RJMP` (Relative JuMP) which uses a 12-bit signed PC-relative offset.)

Data memory can be addressed directly (16-bit address), indirectly via some specific registers (data pairs) or indirectly with an offset. It is also possible to do pre-decrement or post-increment addressing. (Read that carefully!)

Indirect loads can be made from the program memory (e.g. to implement look-up tables).

Registers also provide aliases for a certain subset of the address space.

Arithmetic and logical operations operate on ‘registers’, sometimes with immediate (literal) values substituting for one operand. One operand is always the result register. Sometimes registers act as pairs (e.g. to form 16-bit quantities for addressing). There are some restrictions on some operations with some registers. The ‘conventional’ set of operations {`ADD`, `SUB`, `CP`, `MUL`, `AND`, `OR`, `EOR`} is supported.

Decisions use conditional branches or bit-test-and-conditional skip (the following instruction).

`CALL` stacks the PC in data memory (2 bytes; 3 in some devices) and jumps in program memory to an absolute address.

## AVR instruction set

The instruction set is fundamentally load-store with data operations restricted to registers. A ‘standard’ set of arithmetic/logical operations is provided which includes multiplication (two 8-bit values to a 16-bit result, placed in the R1:R0 register pair) and single bit shifts; division is not implemented. There are some explicit **bit manipulation** instructions; these are sometimes included on such small microcontrollers which are designed for use with LEDs, switches etc.

**Example instruction encodings:**

```
               15                                  0
ADD  Rd, Rr   | 000011  |r| ddddd  | rrrr |
SUB  Rd, Rr   | 000110  |r| ddddd  | rrrr |
LDI  Rd, K    | 1110 | KKKK | dddd | KKKK |R16-R31 only
LD   Rd, X    |  1011000   | ddddd | 1100 |
SBI  A, b     |  10011010  | AAAAA | bbb  |⎫I/O registers
SBIC A, b     |  10011001  | AAAAA | bbb  |⎭0-31 only
CALL  k       |  1001010   | kkkkk | 111|k|2 word instr.
              |    kkkkkkkkkkkkkkkk       |
RCALL k       | 1101 |   kkkkkkkkkkkk     |
RET           |    1001010100001000       |
```

Note features such as:

- ❑ restrictions on register ranges
- ❑ non-adjacent bitfields being grouped together
  (e.g. the 5-bit Rr specifier in `ADD`, the 8-bit K constant in `LDI`)
- ❑ bit manipulation instructions
  (e.g. **S**et **B**it in **I**/o reg. – has 5-bit reg. address and 3-bit bit position)
- ❑ skip instructions (e.g. **S**kip if **B**it in **I**/o is **C**lear)
  ‘skip’s allow a predicate on the next instruction only - a simple (short) form of conditional execution; the potentially skipped instruction could be a branch

Also, observe the choice of assembler syntax: e.g.

- ❑ using the mnemonic (‘`LDI`’) for immediate rather than a ‘#’ mode
- ❑ the implied indirection in ‘`LD`’ (no ‘[‘, ‘]’)

The ISA datasheet is available on line from Atmel. A full(?) instruction map is on the appropriate Wikipedia page.

# Summary

- Not all ISAs are ‘the same’ but most have a lot in common
  - the choice of mnemonics/syntax varies slightly – this is independent of the ISA most assembly languages list operands
- ISAs are extended over time but the basis remains fixed
  - implementations tend to become increasingly complicated
- There are a number of other techniques which have been (and often still are) employed, some of which might be new to you:
  - Variable length instructions
  - Separate I/O space
  - Harvard architecture
  - Short-form ‘addresses’
  - …

## Other stuff

The object of this lecture was to illustrate some of the similarities and differences in microprocessor ISAs. Hopefully some of the material is familiar and there have also been some new concepts and instructions.

It is impossible to cover everything everyone has tried in this space: instead it should give a picture of the scope and, when you do encounter something new or unusual, you will not now be too surprised.

Here are a couple more concepts if you want to investigate further:

- Not all ISA’s have status flags: some use a (specified?) register state.
  – already seen in MU0
- Registers may be indexed: e.g. SPARC uses ‘register windows’ where (some of) the programme’s registers change physical definition on method entry/exit.
  This allows the (partial) implementation of a stack without using memory.

### Designing an ISA

- Inventing a new ISA is easy; you should now be able to do your own!
- Inventing a *good* new ISA is harder.
  - A large amount of simulation is needed to determine how often each proposed instruction might be used in ‘real’ code.
  - This requires compilers and ‘representative’ benchmarks.
- Getting a new ISA adopted is harder still.
  - The processor is easy compared to the software support needed.
  - Legacy (object) code is a great deterrent to innovation.
- Many potentially good ISAs are now obsolete or at least hard to find
  - M68000, Alpha, PowerPC, SPARC, Itanium(?) …

## Assembly languages

Although it is hard to discuss one without the other, the assembly language is independent from instruction set. The choice of assembler syntax is a software issue and it is not uncommon to encounter (usually only slightly) different representations of mnemonics for the same ISA.

Between ISAs, assembly language is normally recognisable (addition is typically ‘`ADD`’!) but details will vary. Copying the contents of one register into another is typically specified by ‘`MOV`’ or ‘`MOVE`’ but sometimes ‘`LD`’ or ‘`LOAD`’; this makes no difference to the actual operation.

As with most imperative languages, addressing modes are usually read right-to-left with the destination being the first (leftmost) specified field. This is not universal: some assembly languages work left-to-right. (In practice this is only briefly confusing, at the point of changing from one to another.) Note also that not ever ISA is internally consistent: ARM/Stump loads and stores have the same syntax (which makes the assembler simpler to write, especially as there is only one bit difference in their op-codes) but move data in opposite directions.

Syntax also varies as to *how* operands are specified. For immediate (literal) operands a ‘`#`’ prefix is common although 8080 and x86 do without whilst AVR codes this into the mnemonic (e.g. ‘`SUBI`’ as opposed to ‘`SUB`’). Possibly the commonest specifier for indirection of addresses bracketing ‘[‘, ‘]’; ‘(‘, ‘)’ are sometimes used but are confusable with expression parentheses. Another alternative is a ‘`@`’ prefix which makes some logical sense: ‘`@R5`’ would be read as ‘at register five’.