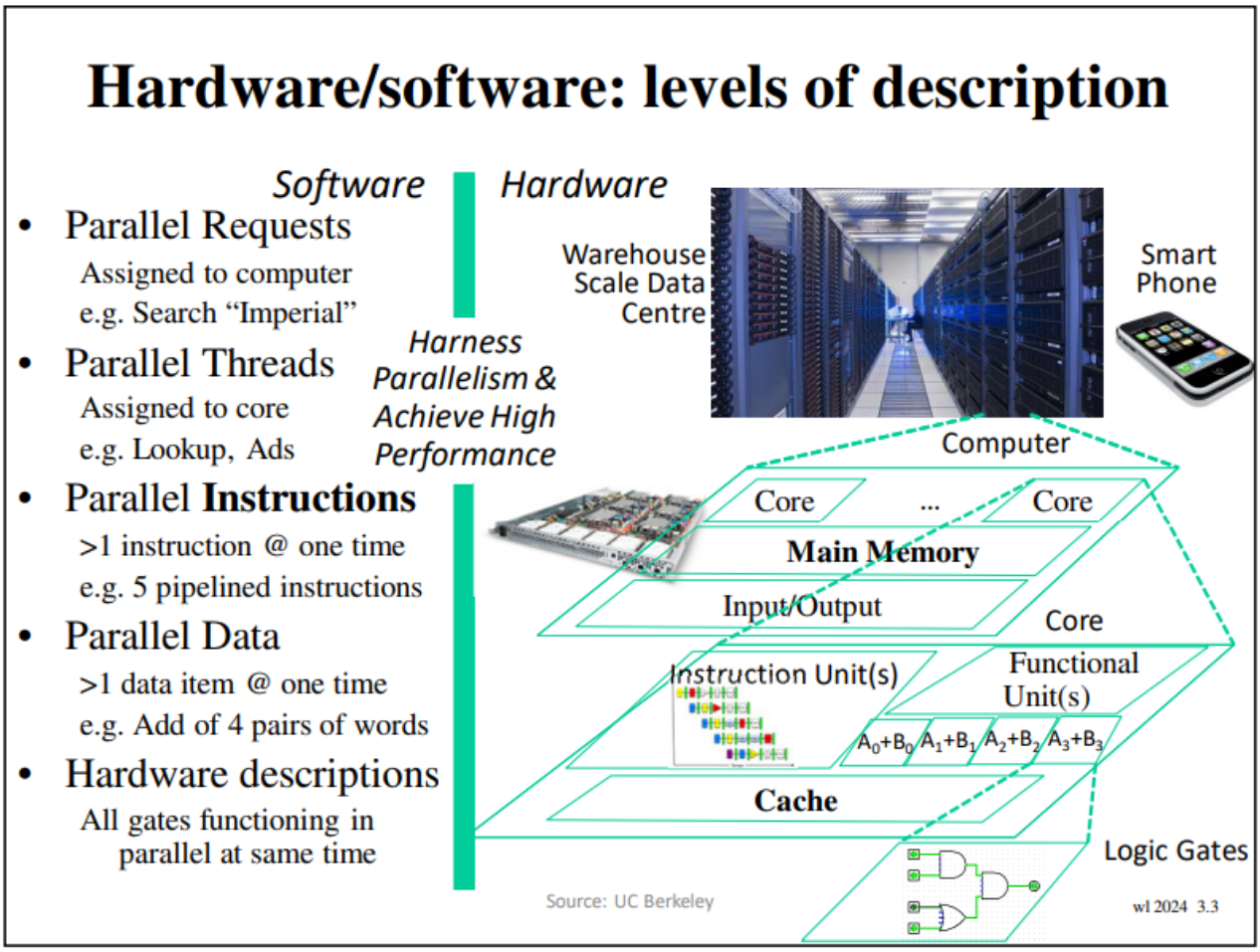
Lecture 3, Memory Addressing and Architecture Evolution

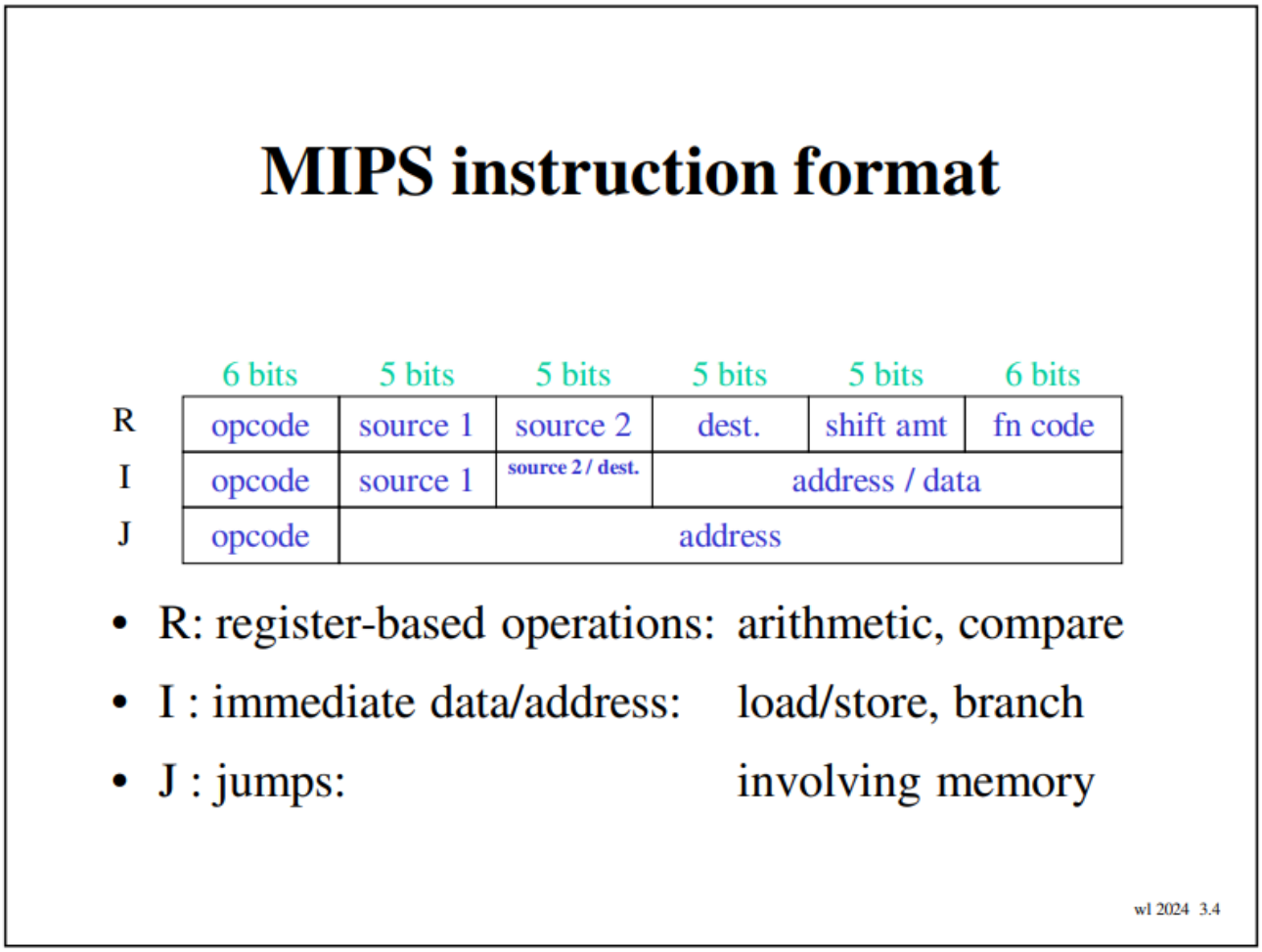
Parallelism



- Parallel Requests
  - Assigned to computer
- Parallel Threads
  - Assigned to core
- Parallel Instructions (more than 1 instruction at one time)
- Parallel Data (more than 1 data item at one time)
- Hardware descriptions
  - All gates functioning in parallel at same time

From servers in warehouse scale data centres to embedded processors in your smart phone, almost all current computers follow a similar architecture. An effective way of understanding this architecture in supporting, for example,

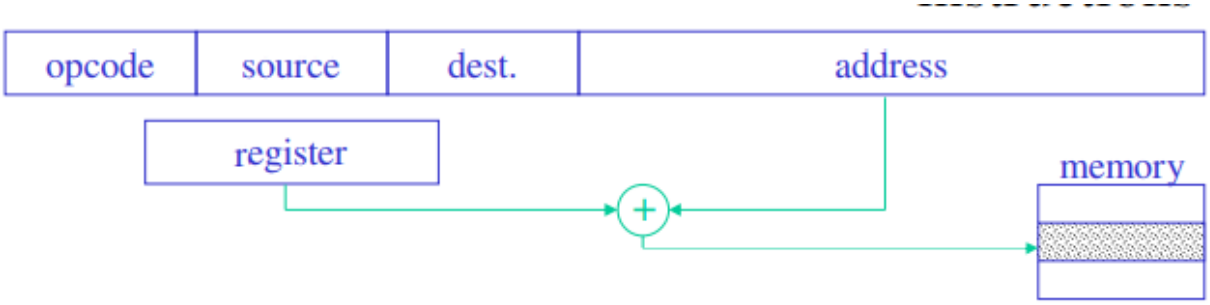
parallelism is to look at the different levels of description for hardware and software. From the hardware perspective, there are multiple cores accessing instructions and data in the main memory, while each core having multiple instruction and functional units accessing data in their cache. From the software perspective, parallel requests result in parallel threads executing instructions accessing data in parallel. The key is to understand that the instruction set architecture enables hardware resources to be accessible to software.



A quick summary of the three types of MIPS instructions. Note that while R type focuses on register addressing, both I type and J type would involve memory access.

MIPS addressing modes

- register addressing: data in registers
- immediate addressing: data in instruction itself (I-type)
- base addressing: data in memory, e.g. load/store instructions



- PC-relative: like base addressing with register replaced by the PC

Addressing modes are ways that a processor can access various types of memory including registers. Different instructions have different ways of accessing data, depending on whether the data are stored in registers, in the instruction, or in memory

MIPS and 68000

- MIPS is typical RISC
- 68000 is typical CISC
- differences: reflect technology advances
  - 68000: fewer registers, less regular
- differences: reflect different views
  - CISC: reduce “semantic gap” due to high-level languages
    - CISC Assembly language is closer to high-level language than RISC

Compare 68000 and MIPS

- 8 data registers, 8 address registers
- 12 addressing modes (`data reg dir` , `addr reg dir/indir`)
- limited number of arithmetic instructions (operate directly on address registers)
- speed (68040) :
  - benchmark SPECint92: 21 (4.2 times slower)
  - SPECfp92 : 15 (6.5 times slower) (than MIPS R4400)
- cost: $233 (4.7 times cheaper than R4400)
- cost effectiveness?
  - MIPS is 4.2 times faster for integers

- 6.5 times faster for floats
- So MIPS is less effective for integers but more effective for floats

If performance increases, cost increases too.

MIPS costs 4.7 times more than the 68000 – it runs integers for only 4.2 times so less effective, while it runs floats 6.5 times faster, so more cost effective.

Addressing comparison

- 68000 has auto-increment mode

```
        (a0)+                   M[a0] before a0 = a0 + 4
so  add.l d0  (a0)+             d0 = d0 + M[a0], a0 = a0+4
takes 16 bits
```

- MIPS require 3 instructions

```
lw   $9,   0, ($7)      # reg9 = M[reg7]
add  $7,  $7,  4        # reg7 = reg7 + 4
add  $8,  $8,  $9       # reg8 = reg8 + reg9
```

- Each instruction has 32 bits, so three would take 96 bits

This example shows the drawbacks of RISC in general and MIPS in particular: in this case MIPS has 6 times more instructions than the 68000.

Embedded processors comparison

Feature	AMCC PPC 440GX	Broadcom BCM1250	Cavium Octeon NSP	IBM PPC 750GX	Freescale MPC7448	Freescale MPC8560	PMC-Sierra RM9000x2GL
Architecture	PowerPC (Book E)	SiByte MIPS64	MIPS64-R2 (cnMIPS64)	PowerPC G3	PowerPC e600 (G4+)	PowerPC e500 PowerQuicc III	Enhanced MIPS64
CPU Cores	1	2	1–16	1	1	1	2
Core Freq (MHz)	533–800	600-1,000	300–600	733–1,100	600–1,700	667–1,000	800–1,000
DRAM Bus Freq	166MHz	Up to 400MHz	Up to 800MHz	Up to 200MHz	133–200MHz	333MHz	200MHz
L1 Cache (I/D)	32K/32K	32K/32K	32K/8K	32K/32K	32K/32K	32K/32K	16K/16K
L2 Cache	256K	512K	Up to 1MB	1MB	1MB	256K	256K per CPU
FPU	—	Yes	—	Yes	Yes	Yes	Yes
ALU Pipeline	7 stages	9 stages	5 stages	4 stages	7 stages	7 stages	7 stages
Superscalar	2-way	4-way	2-way	2-way	4-way	2-way	2-way
Special Features	GbE, TCP/IP h/w, PCI-X, I^2O msg	3xGbE, PCI, HyperT, PCI, 2xDDR	SPI-4.2, RGMII, security, TOE, reg-ex engines	L2 cache locking, deep bus pipelining	AltiVec, new voltage/freq scaling	RapidIO, 2xGbE, PCI-X, DDR-333, MMU, Book E	HyperT, GbE, PCI, DDR, SysAD
Voltage (core)	1.5V	1.2V	1.0–1.2V	1.45V	1.0–1.3V*	1.2V	1.2V
Power (typical)	4.5W (533MHz)	8W–10W (800MHz)	5W–25W (worst-case)	8.8W (1.0GHz)	<10W* (1.4–1.5GHz)	7.5W (600MHz)	<12W
IC Process	0.13µm	0.13µm	0.13µm	0.13µm SOI	90nm SOI	0.13µm	0.13µm (LV)
Package	CBGA-552	BGA-860	709–1,500 pins	CBGA-292	BGA/LGA-360	FCBGA-733	672–896 pins
Production Availability	Now	Now	Now	Now	Now	Now	Now
Price (10K)	$62 (533MHz)	$300–$400	$20–$750	$105 (1K) (1.0GHz)	$47–$332	$104–$140	$321 (800MHz)

* vendor estimate

Prices for 10000 units.

This table compares various embedded RISC processors, including three based on the MIPS ISA. They are different in the number of cores, the core speed, the amount of L1 and L2 caches, their power consumption, their price, and so on,

Classifying architectures

- based on how memory is addressed
- stack: operands specified implicitly at top of stack

  $C = A+B \rightarrow$ `push A; push B; add; pop C`
- accumulator: one operand in accumulator (one register)

  $C = A+B \rightarrow$ `load A; add B; store C`
- register: explicit operands

  $C = A+B \rightarrow$ `load R1 A; add R2,R1,B; store C,R2`
- register advantages: faster than memory, reduce memory traffic, compiler friendly, improve code density

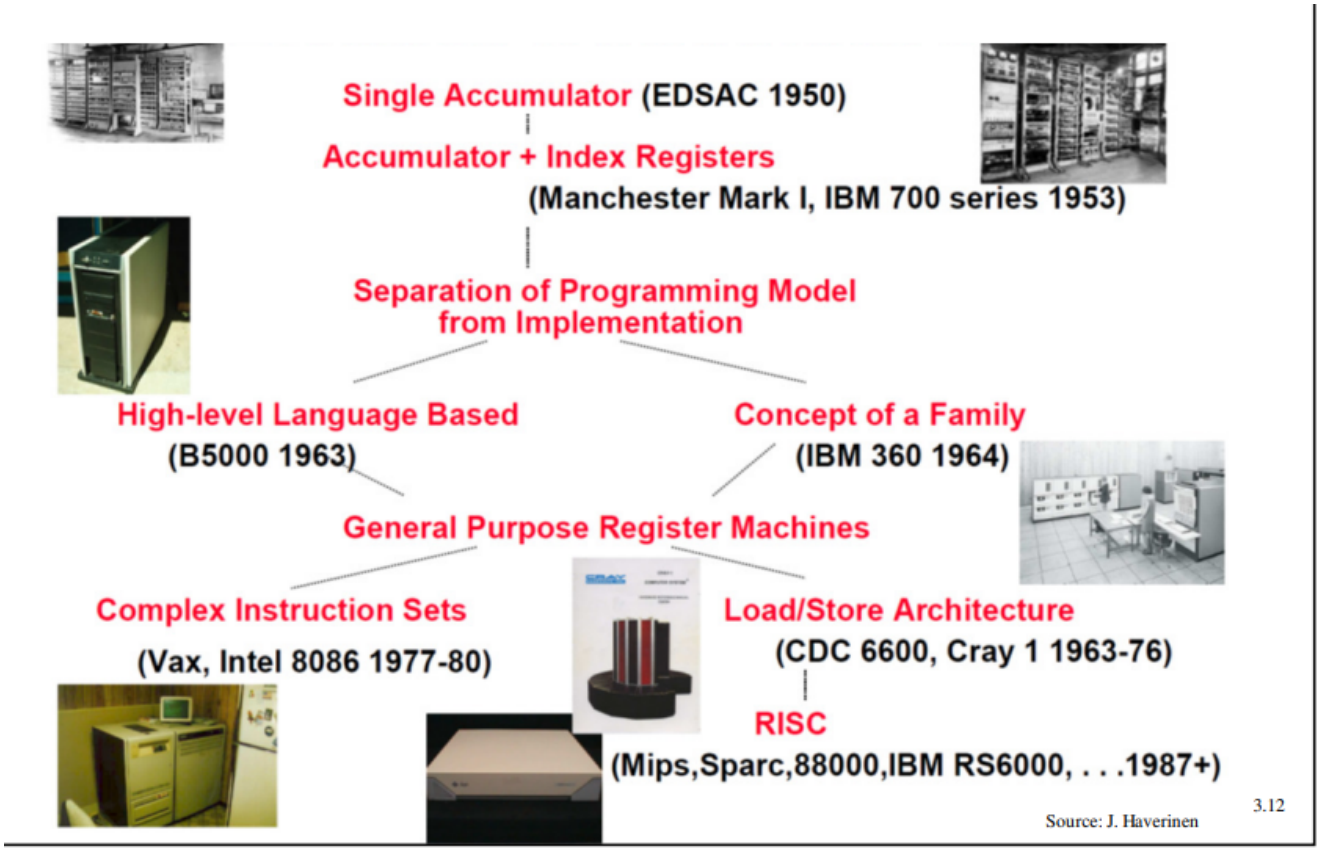
Architectures can be classified depending on how memory is addressed: stack has the most compact instructions since, for example, addresses can be found using the stack pointer so arithmetic instructions do not need addresses of

registers or memory. Instructions involving explicit operands are less compact, but there can be fewer instructions. Accumulator based architectures can be seen as a special case of register based architectures in which there is only one register – called the accumulator.

Comparing architectures

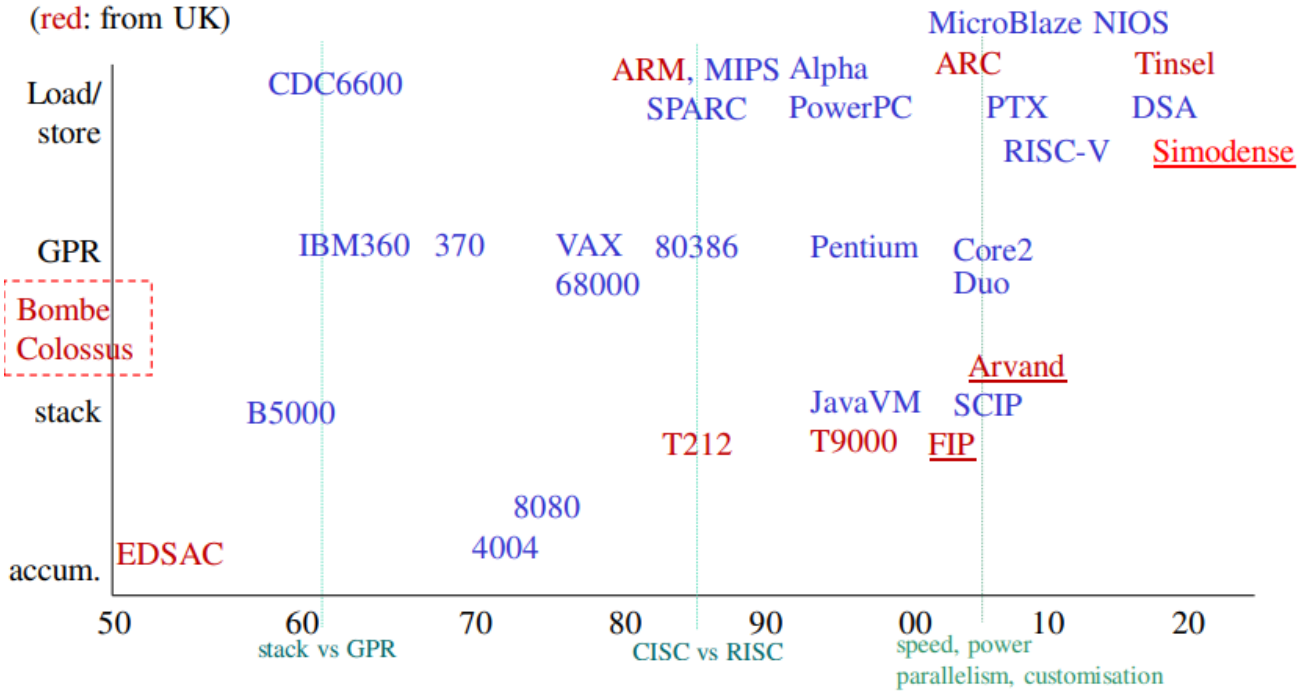
Examples	temporary storage	example	pros	cons
B5500 HP 3000/70	stack	add top pair on stack	simple eval model; dense code	less flexible: no random access; slow if stack in memory
PDP 8 M6809	accumulator	add accum. and memory	min. internal store; short instr.	freq. memory access: slow
VAX MIPS	registers or memory	add 2 registers	general model for code gen., fast reg. access	name all operands; long instr.

Instruction set evolution



This shows how instruction set architectures evolve over the years, starting from the EDSAC in 1950. The early processors were accumulator based, since memory was costly. The first Load/Store architectures were supercomputers, since performance was paramount while cost was less of a concern. As memory gets cheaper, more architectures can afford to have multiple registers, leading to RISC and CISC machines.

Architecture evolution timeline



Criteria: instruction count? code size? performance? parallelism: fixed/customisable? power/energy? evolve with changing requirements, application and environment?

This diagram summaries architecture evolution in the past 70 years. The key moments are indicated: stack vs GPR, CISC vs RISC, and the more recent focus on power consumption and parallelism. The latest development concerns domain specific architectures (DSAs) such as the TPU (see Lecture 1), which are optimised to support a specific application domain such as artificial intelligence. Such domain specific optimisations can be achieved by means of custom instructions; an example is the Simodense processor, a high-performance open-source RISC-V (RV32IM) softcore optimised for exploring custom SIMD instructions designed by my former research student, Dr Philippos Papaphilippou. Two earlier processors were also invented by my former PhD students: FIP (Flexible Instruction Processor) which was used in accelerating the JVM was developed by Dr Shay Ping Seng, while the Arvand processor which was used in accelerating inductive logic programming was developed by Dr Andreas Fidjeland. I was so fortunate to be the PhD supervisor of such brilliant researchers! Many of the latest processors are based on domain specific architectures (DSAs) such as the TPU. Interestingly, some of the earliest computers were also domain specific: examples are Bombe and Colossus, which were used to break encrypted codes by the Enigma and Lorenz machines.

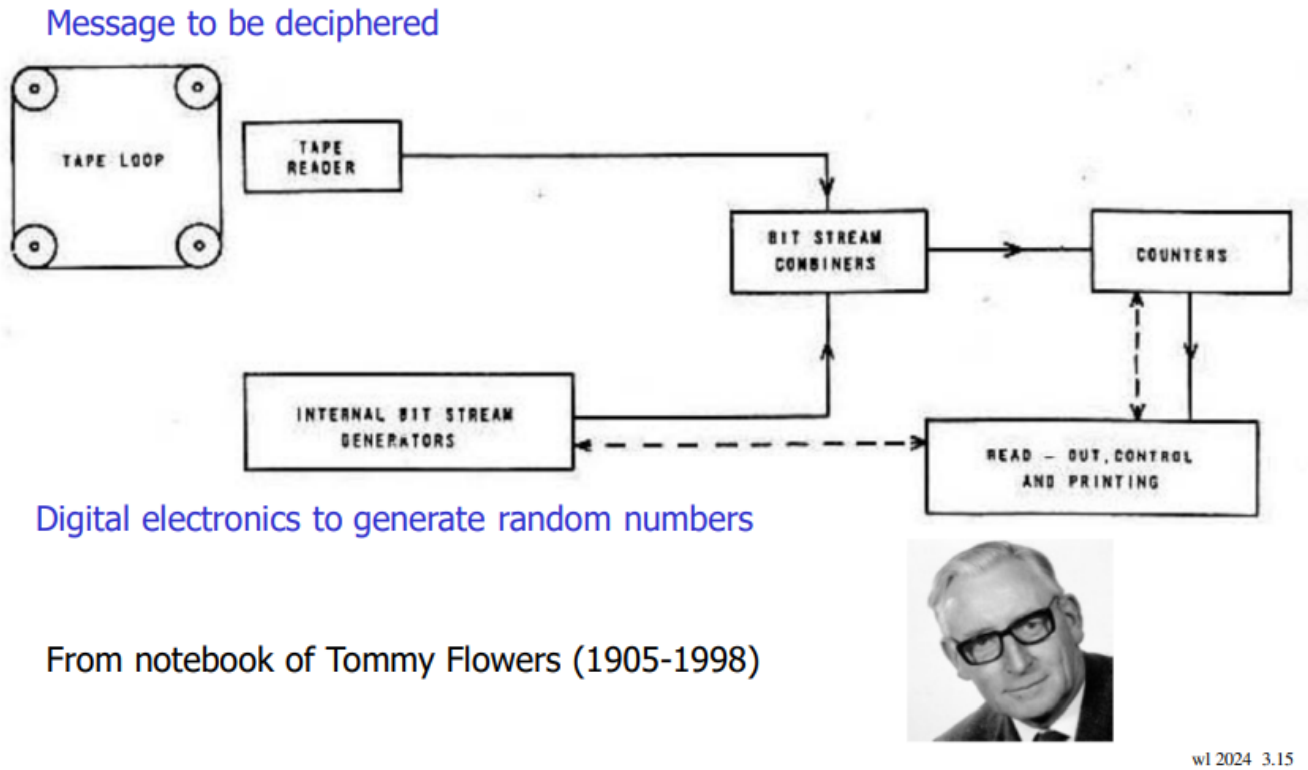
Early 1940s

- Max Newman, Cambridge mathematician

    - tried to automate search for wheel positions of Lorenz, latest Nazi coding machines
- Heath Robinson machine: 2 paper tapes
    - message to be decrypted
    - random numbers for statistical analysis
- synchronising 2 paper tapes was hard
    - slow: up to 2000 characters/second
    - unreliable: answer not always correct
    - prone to catching fire!

The development of the Bombe machine is well known, partly due to the movie “the imitation game”. The Colossus machine is perhaps less famous, but just as important as the Bombe, if not more. Lorenz is a more advanced coding machine than the Enigma. Heath Robinson, the first attempt designed to defeat Lorenz, was slow, unreliable and prone to catching fire. Fortunately Tommy Flowers, an engineer at the General Post Office in Dollis Hill, had a brilliant idea…

Eliminate synchronising paper tapes



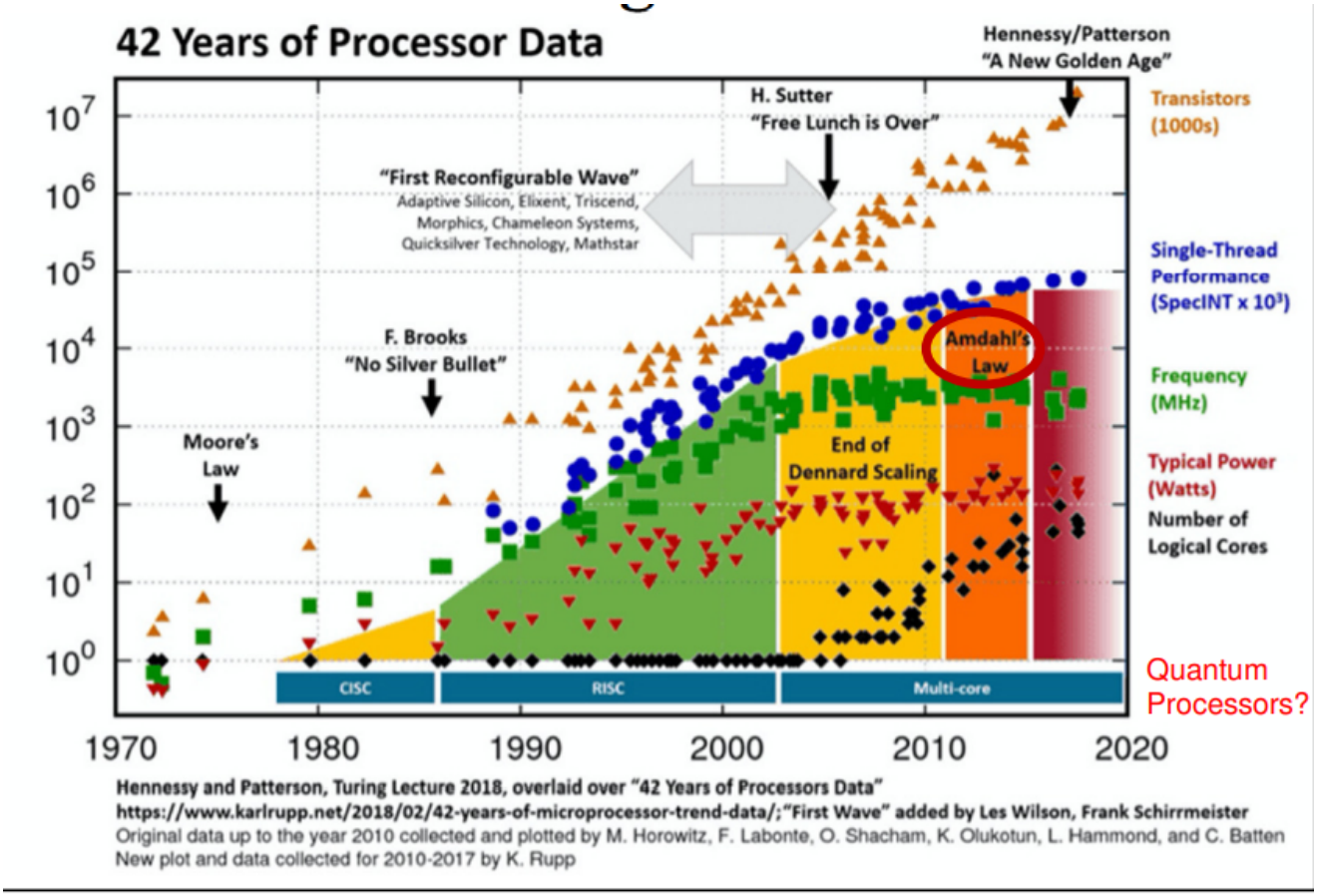
Tommy Flowers invented a new machine that only used one paper tape rather than two. His machine was much faster and more reliable than the Heath Robinson. It was called Colossus. Came in 1944.

Colossus features

- prototype operating end of 1943 (Turing not involved)
  - first domain-specific electronic digital computer
  - had shift registers, branch logic, data storage
  - Mark II installed 5 days before D-Day, June 1944
- parallelism
  - 5 processors, consecutive inputs from one tape
  - 25,000 characters per second, more reliable than Heath Rob
  - load one tape while processing the other tape (pipelining)
- enabling technology
  - 1500-2400 vacuum tubes
  - Flowers worked out how they can operate reliably
- Lower risk of fire

The Colossus was often regarded to be the first domain-specific electronic digital computer, which played a key role in winning World War II. It pioneered various features in today's computer architectures, such as branch logic and parallelism. Colossus was enabled by the use of thousands of vacuum tubes, which were first thought to be too unreliable to use. Tommy Flowers worked out how they could work reliably, and he even put his own money in the Colossus project…

The Big Picture



Lets get back to look at the trend of processors in the last 40-50 years. This is a diagram I show before, now with more details. Look up the significant events to find out more: Moore's Law, F. Brooks "No Silver Bullet" article, the First Reconfigurable Wave… I know various people involved in the companies mentioned, such as Adaptive Silicon and Quicksilver…

But what is the future? How about quantum processors? Hopefully some of you will become pioneers of next-generation processors… The figure above mentions Amdahl's Law. Lets have a look at what it is.

Custom instructions + Amdahl's Law

- Example: repeatedly calculate $x^2 + y^2$
- possible implementations:
  1. add instruction, accumulator or load-store
  2. add and square instructions, accumulator or load-store
  3. custom `sumsq` instruction: dedicated circuit for $x^2 + y^2$
- Amdahl's Law
  - program with fraction α of runtime T_{old} faster by β times

– $T_{new} = \alpha T_{old}/\beta + (1-\alpha)\, T_{old}$

– 90% of code runtime faster by 100 times: $\alpha=0.9$, $\beta=100$

so $T_{new} = 0.109 T_{old}$ or $T_{old} = 9.17 T_{new}$

wl 2024 3.20

A domain specific architecture can be obtained by having special instructions, called custom instructions, which are customized to a specific domain. Recent RISC processors such as RISC-V are designed with custom instructions in mind, so that they can become domain specific architectures by having different custom instructions. How to design custom instructions? The key issue is that custom instructions, or any optimisation, can typically only improve a fraction of the current runtime. So the larger the fraction the better. Amdahl's Law can be used to quantify this situation. The results can be unexpected: even if we could make 90% of the runtime faster by 100 times, the overall speed improvement is less than 9.2 times!

Summary

- computer architecture

  = instruction set architecture + machine organisation

- $CPI = \sum (CPI_i \times \text{instr. count}_i) / (\sum \text{instr. count}_i)$
- execution time equation:

$$\text{exe. time} = \text{instr. count} \times CPI \times \text{cycle time}$$

- CISC: $\downarrow$ instr. count, $\downarrow$ code size, $\uparrow$ CPI, $\uparrow$ cycle time

  RISC: $\uparrow$ instr. count, $\uparrow$ code size, $\downarrow$ CPI, $\downarrow$ cycle time

- Amdahl's Law: $T_{new} = \alpha T_{old}/\beta + (1-\alpha)\, T_{old}$

wl 20

A summary of the important materials covered so far, including two key equations. It also shows how the execution time equation can reveal the secrets of RISC and CISC architectures, and how they manage to improve execution time.