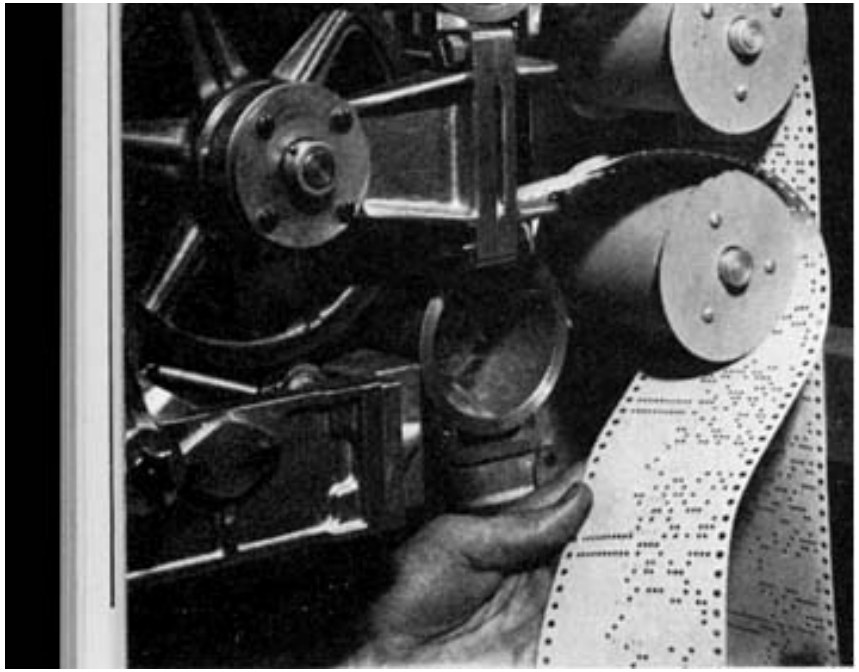
- Agenda:
  1. bridge gap between register machines and real-world micro-processors, and
  2. introduce various design *paradigms*,

  via two case-studies.

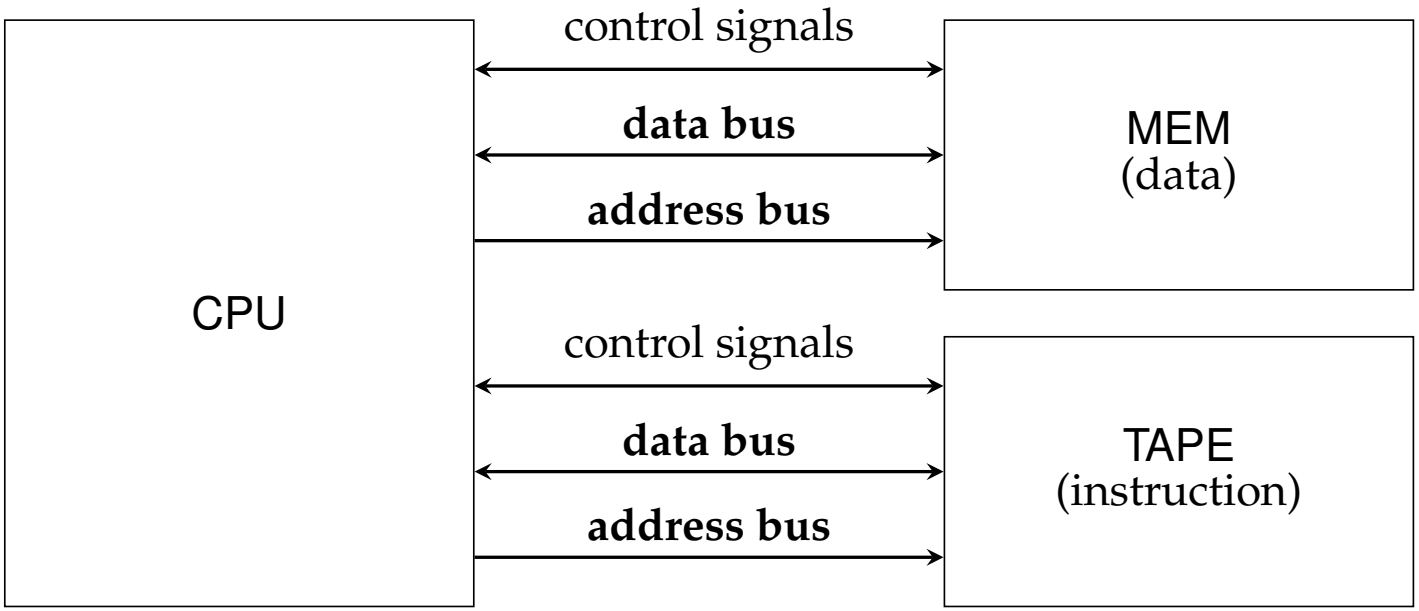
Part 1: ASCC: a Harvard architecture (1)

- Example: **Automatic Sequence Controlled Calculator (ASCC)** [3].
  - Designed by Aiken and IBM; installed at Harvard University circa 1944,
  - 23.0m^2 footprint; 4.3t weight,
  - 72-element, 23-digit memory (i.e., decimal representation),
  - upto $\sim$ 3 operations per-second,
  - programs read from paper tape via a tape reader.

- Example: **Automatic Sequence Controlled Calculator (ASCC)** [3].



- Example: **Automatic Sequence Controlled Calculator (ASCC)** [3].
  - consider an imaginary, ASCC-*like* design which
    - is similar to an accumulator machine, so has an accumulator called A,
    - uses a 6-digit decimal representation of values,
    - understands instructions from some set,
  - this design is an example of a **Harvard architecture**,
  - i.e., there are two, *separate* data and instruction memories:



  - the program, which is a sequence of instructions, is stored on TAPE.

Example (instruction set)

- nop, i.e., do nothing.
- halt, i.e., halt or stop execution.
- $\mathsf{A} \leftarrow n$, i.e., load the number n into the accumulator A.
- $\mathsf{MEM}[n] \leftarrow \mathsf{A}$, i.e., store the number in accumulator A into address n of the memory.
- $\mathsf{A} \leftarrow \mathsf{MEM}[n]$, i.e., load the number in address n in memory into the accumulator A.
- $\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[n]$, i.e., add the number in address n of the memory to the accumulator A and store the result back in the accumulator.
- $\mathsf{A} \leftarrow \mathsf{A} - \mathsf{MEM}[n]$, i.e., subtract the number in address n of the memory from the accumulator A and store the result back in the accumulator.
- $\mathsf{A} \leftarrow \mathsf{A} \oplus \mathsf{MEM}[n]$, i.e., XOR the number in address n of the memory with the accumulator A and store the result back in the accumulator.

Algorithm (fetch-decode-execute cycle)

1. Encode a program P onto paper tape; load this tape into the tape reader, then zero A and start the computer.
2. Using the tape reader, fetch the next instruction in the program and store it in IR.
3. If
   - 3.1 IR $=\perp$ (i.e., an invalid instruction is encountered), or
   - 3.2 IR = halt (i.e., the program halts normally)

   then halt the computer, otherwise execute IR (i.e., perform the operation it specifies).
4. Repeat from step 2.

Example (compute 10 + 20)

CPU		
state	=	reset
IR	=	
A	=	0

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
IR	=	A ← MEM[4]
A	=	0

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	execute
IR	=	A ← MEM[4]
A	=	10

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
IR	=	A ← A + MEM[5]
A	=	10

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	execute
IR	=	A ← A + MEM[5]
A	=	30

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
IR	=	MEM[6] ← A
A	=	30

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	execute
IR	=	MEM[6] ← A
A	=	30

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	30
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
IR	=	halt
A	=	30

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	30
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

Example (compute 10 + 20)

CPU		
state	=	execute
IR	=	halt
A	=	30

MEM	
Address	Value
0	0
1	0
2	0
3	0
4	10
5	20
6	30
7	0

TAPE	
Address	Semantics
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	halt
4	nop
5	nop
6	nop
7	nop

An Aside: how imaginary is imaginary?

- Example:
  - Released circa 1980, the **Intel 8051** [4] is a
    - 8-bit (i.e., has an 8-bit ALU and accumulator A),
    - Harvard architecture,

    micro-controller, with upto
    - 256B of internal data memory (or IRAM),
    - 64kB of external data memory (or XRAM), and
    - 64kB of program memory (or PRAM).
  - The 8051 instruction set isn't *too* far off what we have, e.g.,

$$
\begin{array}{lcl}
\texttt{NOP} & \mapsto & \\
\texttt{MOV x} & \mapsto & \mathsf{A} \leftarrow \texttt{x} \\
\texttt{MOV y} & \mapsto & \mathsf{A} \leftarrow \mathsf{MEM}[\texttt{y}] \\
\texttt{MOV y} & \mapsto & \mathsf{MEM}[\texttt{y}] \leftarrow \mathsf{A} \\
\texttt{ADDC y} & \mapsto & \mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[\texttt{y}] + \text{ carry} \\
\texttt{SUBB y} & \mapsto & \mathsf{A} \leftarrow \mathsf{A} - \mathsf{MEM}[\texttt{y}] - \text{ carry} \\
\texttt{XRL y} & \mapsto & \mathsf{A} \leftarrow \mathsf{A} \oplus \mathsf{MEM}[\texttt{y}]
\end{array}
$$

  where
  - `x` is an 8-bit value, and `y` is an address in IRAM, and
  - there are a *variety* of different versions of `MOV`.

Part 2: EDVAC: a Princeton (aka. von Neumann) architecture (1)

Hang on, why have two *separate* memories [13]: we can have one *unified* memory, and use it to store both data *and* data that represents instructions (i.e., encoded instructions).

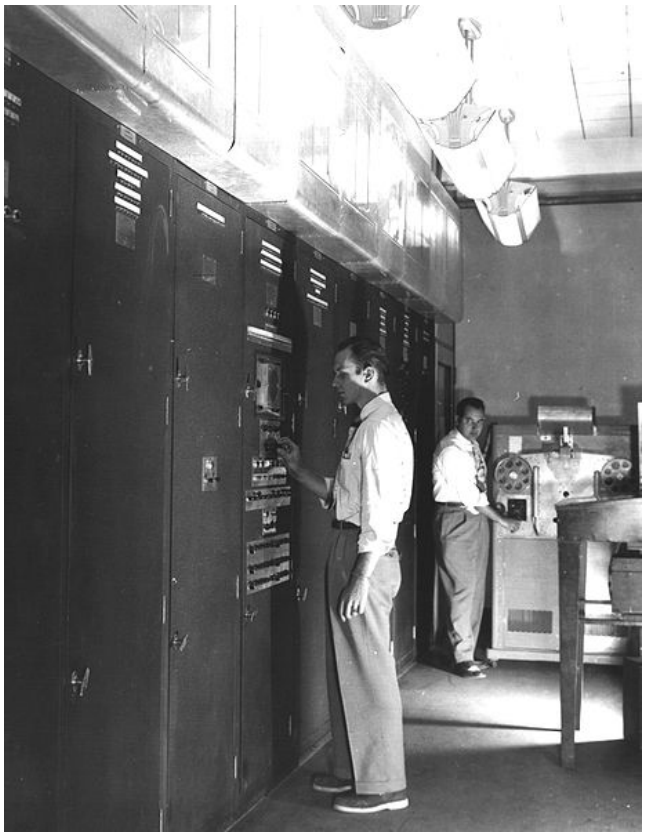
https://en.wikipedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif

Part 2: EDVAC: a Princeton (aka. von Neumann) architecture (2)

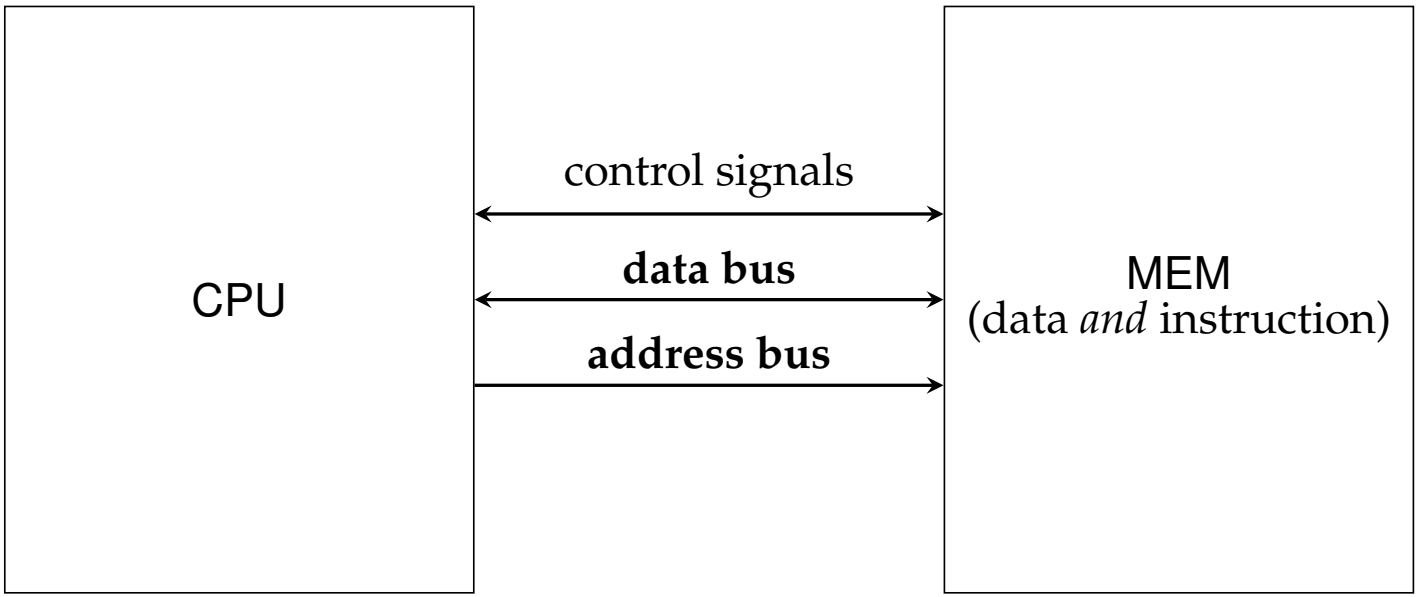
- Example: **Electronic Discrete Variable Automatic Computer (EDVAC)** [1].
  - Designed by Mauchly and Eckert; installed at US-based BRL circa 1949,
  - 45.0m^2 footprint; 7.8t weight,
  - 1000-element, 44-bit memory (i.e., binary representation),
  - upto $\sim$ 1160 operations per-second,
  - programs read from magnetic tape into memory via a tape reader.

Part 2: EDVAC: a Princeton (aka. von Neumann) architecture (2)

- Example: **Electronic Discrete Variable Automatic Computer (EDVAC)** [1].



- Example: **Electronic Discrete Variable Automatic Computer (EDVAC)** [1].
  - consider an imaginary, EDVAC-*like* design which
    - inherits the previous ASCC-like design,
    - adds a program counter called PC,
    - updates the instruction set, e.g., to allow control of PC and define an instruction encoding,
    - updates the fetch-decode-execute cycle to match.
  - this design is an example of a **Princeton architecture** (aka. **von Neumann architecture**),
  - i.e., there is one, *unified* data and instruction memory:



  - the program, which is a sequence of instructions, is stored on MEM.

Example (instruction set)

- 00*nnnn* means nop.
- 10*nnnn* means halt.
- 20*nnnn* means $\mathsf{A} \leftarrow n$.
- 21*nnnn* means $\mathsf{MEM}[n] \leftarrow \mathsf{A}$.
- 22*nnnn* means $\mathsf{A} \leftarrow \mathsf{MEM}[n]$.
- 30*nnnn* means $\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[n]$.
- 31*nnnn* means $\mathsf{A} \leftarrow \mathsf{A} - \mathsf{MEM}[n]$.
- 32*nnnn* means $\mathsf{A} \leftarrow \mathsf{A} \oplus \mathsf{MEM}[n]$.
- 40*nnnn* means $\mathsf{PC} \leftarrow n$.
- 41*nnnn* means $\mathsf{PC} \leftarrow n$ iff. $\mathsf{A} = 0$.
- 42*nnnn* means $\mathsf{PC} \leftarrow n$ iff. $\mathsf{A} \neq 0$.

Algorithm (fetch-decode-execute cycle)

1. Encode a program P onto magnetic tape; load this tape into memory using the tape reader, then zero A and PC and start the computer.
2. From the address in PC, fetch the next instruction in the program and store it in IR.
3. Increment PC so it points to the next instruction.
4. If
   - 4.1 IR $=\perp$ (i.e., an invalid instruction is encountered), or
   - 4.2 IR = halt (i.e., the program halts normally)

   then halt the computer, otherwise execute IR (i.e., perform the operation it specifies).
5. Repeat from step 2.

Example (compute 10 + 20)

CPU		
state	=	reset
PC	=	0
IR	=	
	=	
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
PC	=	0
IR	=	220004
	=	
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	decode
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	execute
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
PC	=	1
IR	=	300005
	=	
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	decode
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	execute
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
PC	=	2
IR	=	210006
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	decode
PC	=	3
IR	=	210006
	=	MEM[6] ← A
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	execute
PC	=	3
IR	=	210006
	=	MEM[6] ← A
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	30	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	fetch
PC	=	3
IR	=	100000
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	30	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	decode
PC	=	4
IR	=	100000
	=	halt
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	30	nop
7	0	nop

Example (compute 10 + 20)

CPU		
state	=	execute
PC	=	4
IR	=	100000
	=	halt
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	halt
4	10	nop
5	20	nop
6	30	nop
7	0	nop

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	reset
PC	=	0
IR	=	
	=	
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	fetch
PC	=	0
IR	=	220004
	=	
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	decode
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	execute
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	fetch
PC	=	1
IR	=	300005
	=	
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	decode
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #1 = { 1. we can control PC, *so* 2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	execute
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	fetch
PC	=	2
IR	=	210006
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	decode
PC	=	3
IR	=	210006
	=	MEM[6] ← A
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #1 = { 1. we can control PC, *so* 2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	execute
PC	=	3
IR	=	210006
	=	MEM[6] ← A
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	30	nop
7	0	nop

stored program ⇒ implication #1 = { 1. we can control PC, *so* 2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	fetch
PC	=	3
IR	=	400000
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	30	nop
7	0	nop

stored program ⇒ implication #1 = { 1. we can control PC, *so* 2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	decode
PC	=	4
IR	=	400000
	=	PC ← 0
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	30	nop
7	0	nop

stored program ⇒ implication #1 = { 1. we can control PC, *so* 2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	execute
PC	=	0
IR	=	400000
	=	PC ← 0
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	30	nop
7	0	nop

stored program ⇒ implication #1 = { 1. we can control PC, *so* 2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	fetch
PC	=	0
IR	=	220004
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	30	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	decode
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	30	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including an infinite loop)

CPU		
state	=	execute
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	400000	PC ← 0
4	10	nop
5	20	nop
6	30	nop
7	0	nop

stored program $\Rightarrow$ implication #1 $=$

1. we can control PC, *so*
2. we can, e.g., write loops.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	reset
PC	=	0
IR	=	
	=	
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	fetch
PC	=	0
IR	=	220004
	=	
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	decode
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	0

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	execute
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	fetch
PC	=	1
IR	=	300005
	=	
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	decode
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	10

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	execute
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	fetch
PC	=	2
IR	=	210003
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	decode
PC	=	3
IR	=	210003
	=	MEM[3] ← A
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	100000	halt
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	execute
PC	=	3
IR	=	210003
	=	MEM[3] ← A
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	30	nop
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	fetch
PC	=	3
IR	=	000030
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	30	nop
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	decode
PC	=	4
IR	=	000030
	=	nop
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	30	nop
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	execute
PC	=	4
IR	=	000030
	=	nop
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	30	nop
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	fetch
PC	=	4
IR	=	000010
	=	
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	30	nop
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	decode
PC	=	5
IR	=	000010
	=	nop
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	30	nop
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Example (compute 10 + 20, including programming mistake)

CPU		
state	=	execute
PC	=	5
IR	=	000010
	=	nop
A	=	30

MEM		
Address	Value	Semantics
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210003	MEM[3] ← A
3	30	nop
4	10	nop
5	20	nop
6	0	nop
7	0	nop

stored program ⇒ implication #2 = { 1. data *and* instructions are stored in MEM, *so* 2. we can, e.g., write *self-modifying* code.

Conclusions

Comparison

A **Harvard architecture**

- has high(er) area wrt. use of 2 buses,
- has high(er) bandwidth wrt. use of 2 buses,
- implies less flexible, static memory utilisation,
- can specialise instruction vs. data accesses,
- disallows self-modifying code.

Comparison

A **Princeton architecture** (aka. **von Neumann architecture**)

- has low(er) area wrt. use of 1 bus,
- has low(er) bandwidth wrt. use of 1 bus,
- implies more flexible, dynamic memory utilisation,
- cannot specialise instruction vs. data accesses,
- allows self-modifying code.

Definition

The term **memory wall** [12] refers to a gap between efficiency of instruction execution and memory access. If memory access latency and bandwidth are insufficient, the associated micro-processor may wait (i.e., becomes idle) until instructions and/or data are delivered: the efficiency of memory access will then limit the efficiency of instruction execution, vs. say clock frequency.

Definition

The term **von Neumann bottleneck**, as introduced by Backus [11], is a criticism of

1. the stored-program concept, essentially using a similar argument as the term memory wall [12],
2. the intellectual bottleneck (or limitation) implied by programmers focusing on and optimising for von Neumann architectures.

Conclusions

- Take away points:
  1. Both architectures are of historical, conceptual, *and* practical importance.
  2. Both architectures are viable options, but *both*
     - conceptual constraints, e.g., the memory wall, *and*
     - practical constraints, e.g., area, clock frequency, power consumption,

     highlight the need for intelligent design and implementation.
  3. Variants, e.g., so-called *modified* Harvard architectures, can act as an effective compromise.

Additional Reading

- *Wikipedia: Harvard architecture*. URL: https://en.wikipedia.org/wiki/Harvard_architecture.
- *Wikipedia: von Neumann architecture*. URL: https://en.wikipedia.org/wiki/Von_Neumann_architecture.
- *Wikipedia: Self-modifying code*. URL: https://en.wikipedia.org/wiki/Self-modifying_code.
- *Wikipedia: Modified Harvard architecture*. URL: https://en.wikipedia.org/wiki/Modified_Harvard_architecture.
- D. Page. "Chapter 4: A functional and historical perspective". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009.
- A.S. Tanenbaum and T. Austin. "Section 1.2: Milestones in computer architecture". In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.
- W. Stallings. "Chapter 2: Computer evolution and performance". In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013.

References

[1] *Wikipedia: Electronic Discrete Variable Automatic Computer (EDVAC)*. URL: https://en.wikipedia.org/wiki/EDVAC (see pp. 18–20).

[2] *Wikipedia: Harvard architecture*. URL: https://en.wikipedia.org/wiki/Harvard_architecture (see p. 71).

[3] *Wikipedia: Harvard Mark I*. URL: https://en.wikipedia.org/wiki/Harvard_Mark_I (see pp. 2–4).

[4] *Wikipedia: Intel MCS-51*. URL: https://en.wikipedia.org/wiki/Intel_MCS-51 (see p. 16).

[5] *Wikipedia: Modified Harvard architecture*. URL: https://en.wikipedia.org/wiki/Modified_Harvard_architecture (see p. 71).

[6] *Wikipedia: Self-modifying code*. URL: https://en.wikipedia.org/wiki/Self-modifying_code (see p. 71).

[7] *Wikipedia: von Neumann architecture*. URL: https://en.wikipedia.org/wiki/Von_Neumann_architecture (see p. 71).

[8] D. Page. "Chapter 4: A functional and historical perspective". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009 (see p. 71).

[9] W. Stallings. "Chapter 2: Computer evolution and performance". In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013 (see p. 71).

[10] A.S. Tanenbaum and T. Austin. "Section 1.2: Milestones in computer architecture". In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 71).

[11] J. Backus. "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs". In: *Communications of the ACM (CACM)* 21.8 (1978), pp. 613–641 (see p. 69).

[12] W.A. Wulf and S.A. McKee. "Hitting the memory wall: implications of the obvious". In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24 (see p. 69).

[13] J. von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. 1945 (see p. 17).