

Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
(csdsp@bristol.ac.uk)

September 5, 2025

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
 - ▶ the associated notes page may be pre-populated with extra, written explanation of material covered in lecture(s), plus
 - ▶ anything with a “grey’ed out” header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

► **Agenda: Register Machines (RMs), i.e.,**

1. introduce associated computational model that is “*closely related to what happens in [practical] modern digital computers*” [7, Page 199], then
2. demonstrate how we implement said model in hardware,

noting that

- +ve : theoretically attractive: can be proved equivalent to a Turing machine
- +ve : practically attractive: has clear analogies with, e.g., FSMs
- −ve : some aspects (e.g., infinite sized registers) cannot be realised in practice
- −ve : can be very inefficient

Notes:

- Keep in mind item #54 of Perlis’ Epigrams on Programming [9]: “*[b]eware of the Turing tar-pit in which everything is possible but nothing of interest is easy*”.

Part 1: in theory (1)

Definition

An **instruction** is a description of a computational step.

Definition

A **Register Machine (RM)** is specified by

- a finite number of registers, each of which can store an (infinite) natural number; $R_i \in \mathbb{N}$ denotes the i -th such register for $0 \leq i < r$, and
- a program, consisting of a finite list of instructions of the form

label : body

such that the i -th instruction has label L_i .

Definition

An RM **configuration** is a tuple

$$C = (l, v_0, v_1, \dots, v_{r-1})$$

where

- l is the current label, and
- v_j is the current value stored in register R_j .

Notes:

Part 1: in theory (2)

Definition

A (finite or infinite) computation by some RM is captured by

$$\langle C_0, C_1, C_2, \dots \rangle$$

i.e., a sequence of configurations such that

► for $i = 0$,

$$C_i = (0, v_0, v_1, \dots, v_{r-1})$$

is the **initial configuration** where v_j is the initial value stored in register R_j ,

► for $i > 0$, C_i results from applying the instruction at label L_i to

$$C_{i-1} = (l, v_0, v_1, \dots, v_{r-1}).$$

Definition

A finite computation by some RM is captured by

$$\langle C_0, C_1, C_2, \dots, C_{h-1} \rangle$$

such that in the **halting configuration**

$$C_{h-1} = (l, v_0, v_1, \dots, v_{r-1})$$

the instruction labelled L_l either

- explicitly, or intentionally forces computation to halt, i.e., is a halt instruction, or
- implicitly, or unintentionally forces computation to halt, e.g., causes an error condition.

Notes:

Part 1: in theory (3)

► **Example:** roughly per [7, Chapter 11], consider a case where

1. $r = 4$, i.e., it has 4 registers,
2. each $R_i \in \{0, 1, \dots, 2^4 - 1 = 15\}$, i.e., the registers store (finite) 4-bit values,
3. the set of available (abstract) instruction templates is

$L_i : R_{addr} \leftarrow R_{addr} + 1$ then goto L_{i+1}

$L_i : R_{addr} \leftarrow R_{addr} - 1$ then goto L_{i+1}

$L_i : \text{if } R_{addr} = 0 \text{ then goto } L_{target} \text{ else goto } L_{i+1}$

$L_i : \text{halt}$

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation.

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C0  =  (0, 0, 0, 2, 0)
L0  ~>  if R2 = 0 then goto L5 else goto L1
C1  =  (1, 0, 0, 2, 0)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C1  =  (1, 0, 0, 2, 0)
L1  ~>  R2 ← R2 - 1 then goto L2
C2  =  (2, 0, 0, 1, 0)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C2  =  (2, 0, 0, 1, 0)
L2  ~>  R3 ← R3 + 1 then goto L3
C3  =  (3, 0, 0, 1, 1)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C3  =  (3, 0, 0, 1, 1)
L3  ~>  R1 ← R1 + 1 then goto L4
C4  =  (4, 0, 1, 1, 1)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C4  =  (4, 0, 1, 1, 1)
L4  ~>  if R0 = 0 then goto L0 else goto L5
C5  =  (0, 0, 1, 1, 1)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C5  =  (0, 0, 1, 1, 1)
L0  ~→  if R2 = 0 then goto L5 else goto L1
C6  =  (1, 0, 1, 1, 1)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C6  =  (1, 0, 1, 1, 1)
L1  ~→  R2 ← R2 - 1 then goto L2
C7  =  (2, 0, 1, 0, 1)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C7  =  (2, 0, 1, 0, 1)
L2  ~→  R3 ← R3 + 1 then goto L3
C8  =  (3, 0, 1, 0, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C8  =  (3, 0, 1, 0, 2)
L3  ~→  R1 ← R1 + 1 then goto L4
C9  =  (4, 0, 2, 0, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C9   =   (4, 0, 2, 0, 2)
L4  ~>  if R0 = 0 then goto L0 else goto L5
C10 =   (0, 0, 2, 0, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C10  =   (0, 0, 2, 0, 2)
L0  ~>  if R2 = 0 then goto L5 else goto L1
C11 =   (5, 0, 2, 0, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C11 = (5, 0, 2, 0, 2)
L5  ~> if R1 = 0 then goto L9 else goto L6
C12 = (6, 0, 2, 0, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C12 = (6, 0, 2, 0, 2)
L6  ~> R1 ← R1 - 1 then goto L7
C13 = (7, 0, 1, 0, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C13 = (7, 0, 1, 1, 2)
L7  ~> R2 ← R2 + 1 then goto L8
C14 = (8, 0, 1, 1, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C14 = (8, 0, 1, 1, 2)
L8  ~> if R0 = 0 then goto L5 else goto L9
C15 = (5, 0, 1, 1, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C15 = (5, 0, 1, 1, 2)
L5  ~> if R1 = 0 then goto L9 else goto L6
C16 = (6, 0, 1, 1, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C16 = (6, 0, 1, 1, 2)
L6  ~> R1 ← R1 - 1 then goto L7
C17 = (7, 0, 0, 1, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C17 = (7, 0, 0, 1, 2)
L7  ~> R2 ← R2 + 1 then goto L8
C18 = (8, 0, 0, 2, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C18 = (8, 0, 0, 2, 2)
L8  ~> if R0 = 0 then goto L5 else goto L9
C19 = (5, 0, 0, 2, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C19 = (5, 0, 0, 2, 2)
L5  ~> if R1 = 0 then goto L9 else goto L6
C20 = (9, 0, 0, 2, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C20 = (9, 0, 0, 2, 2)
L9  ~> halt
C21 = (9, 0, 0, 2, 2)
```

Notes:

Part 1: in theory (4)

► Example: now, given

1. a program comprised of (concrete) instruction instances e.g.,

```
L0 : if R2 = 0 then goto L5 else goto L1
L1 : R2 ← R2 - 1 then goto L2
L2 : R3 ← R3 + 1 then goto L3
L3 : R1 ← R1 + 1 then goto L4
L4 : if R0 = 0 then goto L0 else goto L5
L5 : if R1 = 0 then goto L9 else goto L6
L6 : R1 ← R1 - 1 then goto L7
L7 : R2 ← R2 + 1 then goto L8
L8 : if R0 = 0 then goto L5 else goto L9
L9 : halt
```

2. an initial configuration, e.g.,

$$C_0 = (0, 0, 0, 2, 0)$$

we can produce a **trace** of computation:

```
C20 = (9, 0, 0, 2, 2)
L9  ~> halt
C21 = (9, 0, 0, 2, 2)
```

which demonstrates that the program copies R₂ into R₃.

Notes:

Part 1: in theory (5)

► Question: is this the *only* viable example?

► Answer: many **alternatives** exist, stemming, e.g., from

1. different models, e.g.,
 - counter machine,
 - Random-Access Machine (RAM),
 - Random-Access Stored-Program (RASP) machine,
 - ...

Notes:

Part 1: in theory (5)

- ▶ **Question:** is this the *only* viable example?
- ▶ **Answer:** many **alternatives** exist, stemming, e.g., from

2. different instruction set content, e.g., a register machine with or without

$$L_i : R_{addr} \leftarrow 0$$

or “clear” instruction.

Notes:

Part 1: in theory (5)

- ▶ **Question:** is this the *only* viable example?
- ▶ **Answer:** many **alternatives** exist, stemming, e.g., from

3. different instruction set format, e.g.,

- ▶ **register machine** $\approx$ 3-operand model:

- r registers,
- source and destination operands can be specified independently,
- (rough) example:

$$\begin{array}{lll} R_0 \leftarrow 10 & \rightsquigarrow & C_0 = (\quad 0, \quad 0, \quad 0, \quad 0, \quad 0 \quad) \\ R_1 \leftarrow 20 & \rightsquigarrow & C_1 = (\quad 1, \quad 10, \quad 0, \quad 0, \quad 0 \quad) \\ R_2 \leftarrow R_0 + R_1 & \rightsquigarrow & C_2 = (\quad 2, \quad 10, \quad 20, \quad 0, \quad 0 \quad) \\ & & C_3 = (\quad 3, \quad 10, \quad 20, \quad 30, \quad 0 \quad) \end{array}$$

Notes:

Part 1: in theory (5)

- ▶ **Question:** is this the *only* viable example?
- ▶ **Answer:** many **alternatives** exist, stemming, e.g., from

3. different instruction set format, e.g.,

▶ **register machine** $\approx$ 2-operand model:

- r registers,
- operands may need to be reused as source *and* destination,
- (rough) example:

$$\begin{array}{lll} R_0 \leftarrow 10 & \rightsquigarrow & C_0 = (0, 0, 0, 0, 0) \\ R_1 \leftarrow 20 & \rightsquigarrow & C_1 = (1, 10, 0, 0, 0) \\ R_0 \leftarrow R_0 + R_1 & \rightsquigarrow & C_2 = (2, 10, 20, 0, 0) \\ & & C_3 = (3, 30, 20, 0, 0) \end{array}$$

Notes:

Part 1: in theory (5)

- ▶ **Question:** is this the *only* viable example?
- ▶ **Answer:** many **alternatives** exist, stemming, e.g., from

3. different instruction set format, e.g.,

▶ **accumulator machine** $\approx$ 1-operand model:

- may have $r > 1$ register, but there is 1 special-purpose case termed the **accumulator**,
- operations implicitly use accumulator for source and/or destination operands,
- (rough) example:

$$\begin{array}{lll} A \leftarrow 10 & \rightsquigarrow & C_0 = (0, 0, 0, 0, 0) \\ & \rightsquigarrow & C_1 = (1, 10, 0, 0, 0) \\ A \leftarrow A + 20 & \rightsquigarrow & C_2 = (2, 30, 0, 0, 0) \end{array}$$

Notes:

Part 1: in theory (5)

- ▶ **Question:** is this the *only* viable example?
- ▶ **Answer:** many **alternatives** exist, stemming, e.g., from

3. different instruction set format, e.g.,

▶ **stack machine** $\approx$ 0-operand model:

- may have $r > 1$ register, but managed per a **stack** (i.e., FILO-style) policy,
- operations implicitly use stack for source and/or destination operands,
- (rough) example:

	$\leadsto$	$C_0 = ($	0,	0,	0,	0,	0	$)$
push 20	$\leadsto$	$C_1 = ($	1,	20,	0,	0,	0	$)$
push 10	$\leadsto$	$C_2 = ($	2,	10,	20,	0,	0	$)$
add	$\leadsto$	$C_3 = ($	3,	30,	0,	0,	0	$)$
pop	$\leadsto$	$C_4 = ($	4,	0,	0,	0,	0	$)$

Notes:

Part 2: in practice (1)

Design

Definition

Consider a sequence

$$x = \langle x_0, x_1, \dots, x_{n-1} \rangle$$

where, for each $0 \leq i < n$ we have $x_i \in \mathbb{N}$. The associated **Gödel encoding** (or **Gödel numbering**) is

$$\hat{x} = \prod_{i=0}^{i < n} p_i^{x_i} = p_0^{x_0} \cdot p_1^{x_1} \cdots p_{n-1}^{x_{n-1}}$$

where p_i is the i -th prime, i.e., $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, and so on. Due to Euclid's unique prime-factorisation theorem, factoring $\hat{x}$ allows recovery of x .

$\therefore$ we can represent *anything* using elements of $\mathbb{N}$, e.g., per [8, Section VII.A],

- ▶ let 6 represent “0”,
- ▶ let 5 represent “=”, then
- ▶ the logical statement “ $0 = 0$ ” can be represented as

$$2^6 \cdot 3^5 \cdot 5^6 = 243,000,000.$$

Notes:

► Concept:

► One can view

(human-readable) instruction $\simeq$ abstraction of (machine-readable) control information,

i.e.,

instruction	=	information	$\mapsto$	what to do
data	=	information	$\mapsto$	what to do it on/with

Notes:

► Concept:

► Gödel encoding allows numerical representation of *either* form of information, e.g.,

1	$\mapsto$	“compute an addition” if it represents an instruction
1	$\mapsto$	“the integer one” if it represents some data

are different, valid interpretations of the same number.

Notes:

► Concept:

- These facts suggest a strategy: an RM is an FSM in disguise, in the sense that
 1. an RM configuration is an FSM state, and
 2. the RM program determines the FSM transition function,so we could therefore
 - use, e.g., Gödel encoding or variant thereof, to encode instructions into numerical **machine code**,
 - store the machine code in memory,
 - have our implementation decode machine code into appropriate control signals.

Notes:

Definition

The **stored program** concept implies that a mutable program is stored in some form of memory; a **stored program computer** is one whereby instructions in such a program are loaded (or fetched) from memory before execution.

Notes:

Part 2: in practice (4)

Design

Definition

The **Program Counter (PC)** is a special-purpose register that holds the address of the next instruction to be executed.

Definition

The **Instruction Register (IR)** is a special-purpose register that holds the instruction currently being executed.

Notes:

Part 2: in practice (5)

Design

Definition

The **fetch-decode-execute cycle** (aka. **instruction cycle**) is a 3-stage process

1. fetch stage : $\left\{ \begin{array}{ll} \text{1.a.} & \text{load instruction into IR} \quad \mapsto \quad \text{IR} \leftarrow \text{MEM}[\text{PC}] \\ \text{1.b.} & \text{increment PC} \quad \mapsto \quad \text{PC} \leftarrow \text{PC} + 1 \end{array} \right.$
2. decode stage : $\left\{ \begin{array}{l} \text{decide what instruction in IR means, i.e.,} \\ \text{translate IR into control signals which reflect instruction semantics} \end{array} \right.$
3. execute stage : $\left\{ \begin{array}{l} \text{do whatever instruction in IR means, i.e.,} \\ \text{apply instruction semantics} \end{array} \right.$

which describes execution of instructions; in some cases it makes sense to consider a 5-stage process by adding

4. memory access stage : { perform any memory accesses (e.g., loads or stores) required
5. write-back (or commit) stage : { store result(s) stemming from instruction execution (e.g., computation)

i.e., expanding the execute stage to be more precise.

Notes:

Part 2: in practice (6)

Design

- **Design:** specify an instruction encoding, e.g.,

$L_i : R_{addr} \leftarrow R_{addr} + 1 \text{ then goto } L_{i+1}$	$\mapsto$	<table><tr><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td colspan="3">000</td><td colspan="3">addr</td><td colspan="3">0000</td></tr></table>	8	7	6	5	4	3	2	1	0	000			addr			0000		
8	7	6	5	4	3	2	1	0												
000			addr			0000														
$L_i : R_{addr} \leftarrow R_{addr} - 1 \text{ then goto } L_{i+1}$	$\mapsto$	<table><tr><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td colspan="3">001</td><td colspan="3">addr</td><td colspan="3">0000</td></tr></table>	8	7	6	5	4	3	2	1	0	001			addr			0000		
8	7	6	5	4	3	2	1	0												
001			addr			0000														
$L_i : \text{if } R_{addr} = 0 \text{ then goto } L_{target} \text{ else goto } L_{i+1}$	$\mapsto$	<table><tr><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td colspan="3">010</td><td colspan="3">addr</td><td colspan="3">target</td></tr></table>	8	7	6	5	4	3	2	1	0	010			addr			target		
8	7	6	5	4	3	2	1	0												
010			addr			target														
$L_i : \text{halt}$	$\mapsto$	<table><tr><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td colspan="2">011</td><td colspan="2">00</td><td colspan="5">0000</td></tr></table>	8	7	6	5	4	3	2	1	0	011		00		0000				
8	7	6	5	4	3	2	1	0												
011		00		0000																

such that

$$L_i : \text{if } R_2 = 0 \text{ then goto } L_5 \text{ else goto } L_{i+1} \mapsto 010100101_{(2)} = 0A5_{(16)}.$$

Notes:

Part 2: in practice (7)

Design

- **Design:** encode our original program

$L_0 : \text{if } R_2 = 0 \text{ then goto } L_5 \text{ else goto } L_1$	$\mapsto$	$010100101_{(2)}$	$=$	$0A5_{(16)}$
$L_1 : R_2 \leftarrow R_2 - 1 \text{ then goto } L_2$	$\mapsto$	$001100000_{(2)}$	$=$	$060_{(16)}$
$L_2 : R_3 \leftarrow R_3 + 1 \text{ then goto } L_3$	$\mapsto$	$000110000_{(2)}$	$=$	$030_{(16)}$
$L_3 : R_1 \leftarrow R_1 + 1 \text{ then goto } L_4$	$\mapsto$	$000010000_{(2)}$	$=$	$010_{(16)}$
$L_4 : \text{if } R_0 = 0 \text{ then goto } L_0 \text{ else goto } L_5$	$\mapsto$	$010000000_{(2)}$	$=$	$080_{(16)}$
$L_5 : \text{if } R_1 = 0 \text{ then goto } L_9 \text{ else goto } L_6$	$\mapsto$	$010011001_{(2)}$	$=$	$099_{(16)}$
$L_6 : R_1 \leftarrow R_1 - 1 \text{ then goto } L_7$	$\mapsto$	$001010000_{(2)}$	$=$	$050_{(16)}$
$L_7 : R_2 \leftarrow R_2 + 1 \text{ then goto } L_8$	$\mapsto$	$000100000_{(2)}$	$=$	$020_{(16)}$
$L_8 : \text{if } R_0 = 0 \text{ then goto } L_5 \text{ else goto } L_9$	$\mapsto$	$010000101_{(2)}$	$=$	$085_{(16)}$
$L_9 : \text{halt}$	$\mapsto$	$011000000_{(2)}$	$=$	$0C0_{(16)}$

such that

- we use

$$\text{MEM} = \langle 0A5_{(16)}, 060_{(16)}, \dots, 0C0_{(16)} \rangle,$$

- a 10-element memory,
- each $\text{MEM}[i]$ is a 9-bit encoding of the instruction labelled L_i .

Notes:

Part 2: in practice (8)

Design

► Translation:

- we're encoding the instructions as 9-element sequence of bits,
- using a Gödel encoding is too inefficient, so we opt for

$$\begin{aligned}\hat{x} &= x_0 \parallel x_1 \parallel x_2 \\ &\equiv x_0 \cdot 2^0 + x_1 \cdot 2^4 + x_2 \cdot 2^6\end{aligned}$$

so decoding amounts to extraction of contiguous bits from $\hat{x}$, i.e.,

$$\begin{aligned}x_0 &= \hat{x}_{3...0} \equiv (\hat{x} \gg 0) \wedge F_{(16)} \\ x_1 &= \hat{x}_{5...4} \equiv (\hat{x} \gg 4) \wedge 3_{(16)} \\ x_2 &= \hat{x}_{8...6} \equiv (\hat{x} \gg 6) \wedge 7_{(16)}\end{aligned}$$

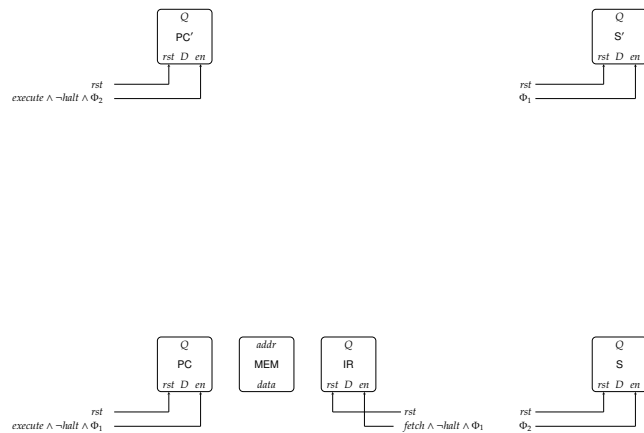
- where a field, e.g., *target*, is unused, we just use zero as a placeholder,
- this approach works, but clearly isn't the *only* one possible.

Notes:

Part 2: in practice (9)

(High-level) implementation: control-path

Circuit

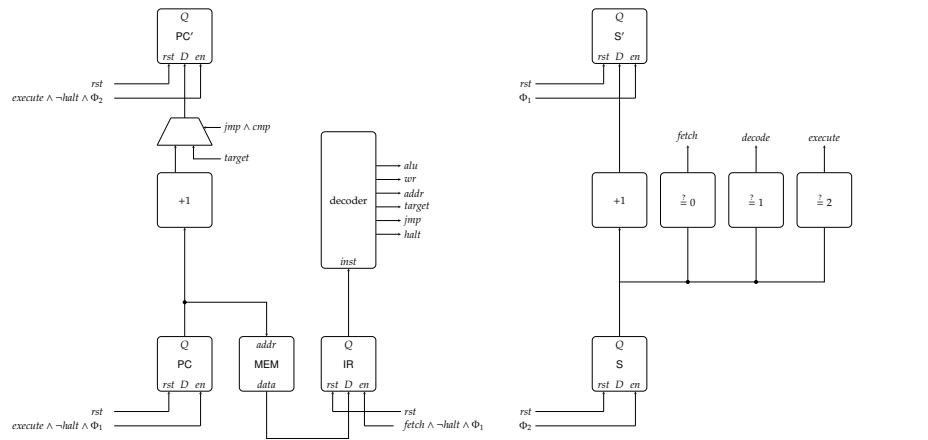


Notes:

Part 2: in practice (9)

(High-level) implementation: control-path

Circuit

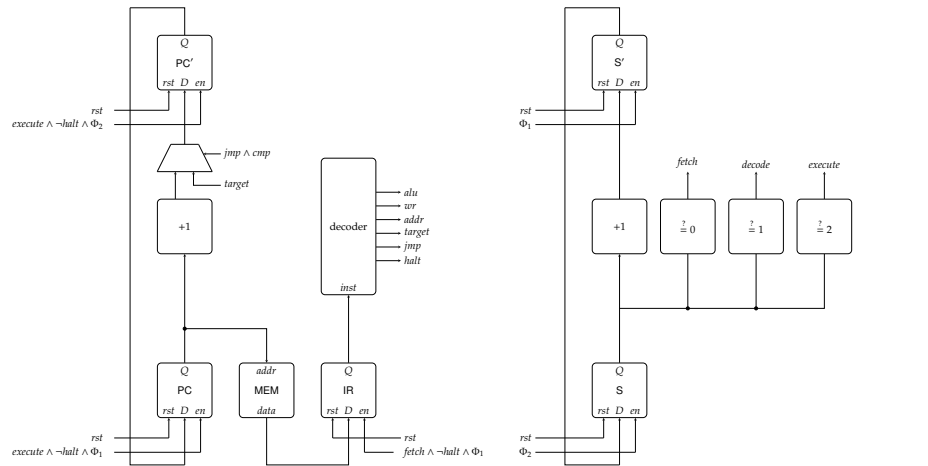


Notes:

Part 2: in practice (9)

(High-level) implementation: control-path

Circuit



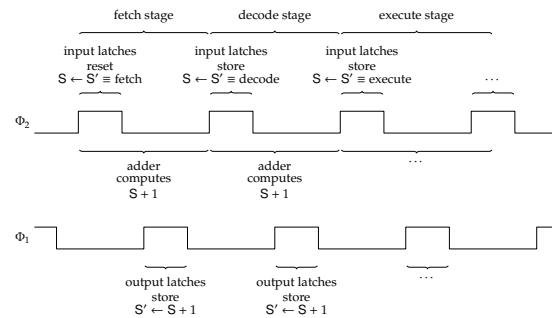
Notes:

► (Too much) **detail** (or, **caveats** to keep in mind):

1. we're using latches (resp. flip-flops) with a dedicated *rst* input (in addition to *en*),
2. we're assuming a cyclic 2-bit counter, so the state *S* steps through values

$0, 1, 2, 3, 0, 1, 2, 3, \dots$,

3. we're not using $S = 3$, so in a sense this is an idle state,
4. we've reversed Φ_1 and Φ_2 for the sequencer, so we generate



to correctly drive update to registers the data- and control-path,

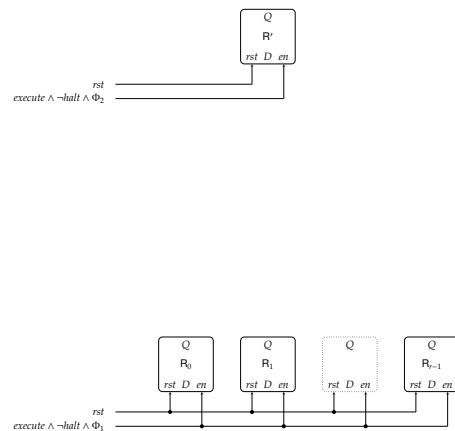
5. we've deviated slightly from the fetch-decode-execute cycle, updating PC in the execute (rather than fetch) stage.

Notes:

Part 2: in practice (11)

(High-level) implementation: data-path

Circuit

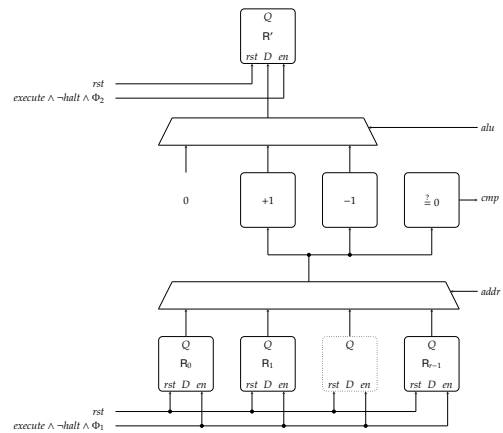


Notes:

Part 2: in practice (11)

(High-level) implementation: data-path

Circuit

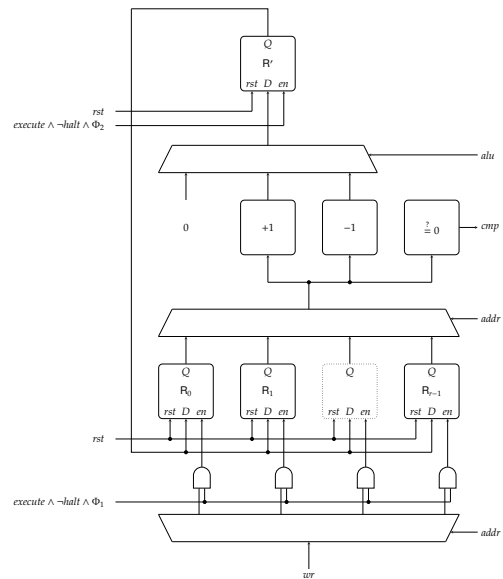


Notes:

Part 2: in practice (11)

(High-level) implementation: data-path

Circuit



Notes:

Conclusions

► Take away points:

1. A central aim here is to demonstrate that, per

combinatorial logic	↔	fixed function, not stateful
sequential logic	↔	fixed function, stateful
FSM	↔	fixed function, stateful
RM	↔	not fixed function, stateful
⋮		
micro-processor	↔	not fixed function, stateful

and although our register machine is *still* limited,

- it has started to exhibit the characteristics of a *real* micro-processor, *and*
- we can *still* reason end-to-end about the implementation.

2. In doing so we've encountered some fundamental concepts, e.g.,

- instruction encoding and decoding,
- the fetch-decode-execute cycle,
- the role of PC and IR in supporting it,
- ...

which we'll revisit and refine.

Notes:

Additional Reading

- *Wikipedia: Register machine*. URL: https://en.wikipedia.org/wiki/Register_machine.
- *Wikipedia: Counter machine*. URL: https://en.wikipedia.org/wiki/Counter_machine.
- *Wikipedia: Random-access machine*. URL: https://en.wikipedia.org/wiki/Random-access_machine.
- *Wikipedia: Random-access stored-program machine*. URL: https://en.wikipedia.org/wiki/Random-access_stored-program_machine.
- *Wikipedia: Gödel numbering*. URL: https://en.wikipedia.org/wiki/G%C3%B6del_numbering.
- *Wikipedia: Machine code*. URL: https://en.wikipedia.org/wiki/Machine_code.

Notes:

References

- [1] *Wikipedia: Counter machine*. URL: https://en.wikipedia.org/wiki/Counter_machine (see p. 107).
- [2] *Wikipedia: Gödel numbering*. URL: https://en.wikipedia.org/wiki/G%C3%B6del_numbering (see p. 107).
- [3] *Wikipedia: Machine code*. URL: https://en.wikipedia.org/wiki/Machine_code (see p. 107).
- [4] *Wikipedia: Random-access machine*. URL: https://en.wikipedia.org/wiki/Random-access_machine (see p. 107).
- [5] *Wikipedia: Random-access stored-program machine*. URL: https://en.wikipedia.org/wiki/Random-access_stored-program_machine (see p. 107).
- [6] *Wikipedia: Register machine*. URL: https://en.wikipedia.org/wiki/Register_machine (see p. 107).
- [7] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967 (see pp. 5, 11).
- [8] E. Nagel and J.R. Newman. *Gödel's Proof*. Routledge, 1958 (see p. 71).
- [9] A.J. Perlis. "Epigrams on programming". In: *ACM SIGPLAN Notices* 17.9 (1982), pp. 7–13 (see p. 6).

Notes: