Intro. to Computer Architecture

Daniel Page

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

January 9, 2018

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 explaination 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:

Continued from last lecture ...

Notes:

"Building Block" Components (1)

Choice

Notes:

- Just two building blocks can support most forms of choice:
  1. a **multiplexer**
     - has m inputs,
     - has 1 output,
     - uses a ($\lceil \log_2(m) \rceil$)-bit control signal input to choose which input is connected to the output,

     while
  2. a **demultiplexer**
     - has 1 input,
     - has m outputs,
     - uses a ($\lceil \log_2(m) \rceil$)-bit control signal input to choose which output is connected to the input,

  noting that
  - the input(s) and output(s) are n-bit, but clearly must match up,
  - the connection made is continuous, since both components are combinatorial.

"Building Block" Components (2)
Choice

- As an analogy, the C `switch` statement

Listing

```
1 switch( c ) {
2   case 0 : r = w; break;
3   case 1 : r = x; break;
4   case 2 : r = y; break;
5   case 3 : r = z; break;
6 }
```

acts similarly to a 4-input multiplexer.

- Likewise,

Listing

```
1 switch( c ) {
2   case 0 : r0 = x; break;
3   case 1 : r1 = x; break;
4   case 2 : r2 = x; break;
5   case 3 : r3 = x; break;
6 }
```

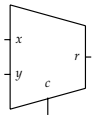
acts similarly to a 4-output demultiplexer.

Notes:

"Building Block" Components (3)
Choice

Definition

The behaviour of a **2-input, 1-bit multiplexer** component



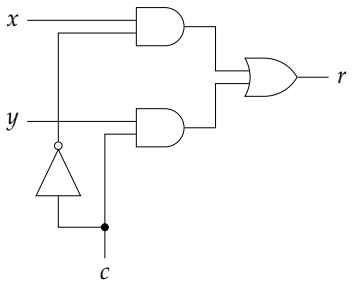
is described by the truth table

MUX2			
c	x	y	r
0	0	**?**	0
0	1	**?**	1
1	**?**	0	0
1	**?**	1	1

which can be used to derive the following implementation:

$$r = (\neg c \wedge x) \vee (c \wedge y)$$

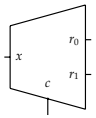
Circuit



Notes:

Definition

The behaviour of a **2-output, 1-bit demultiplexer** component



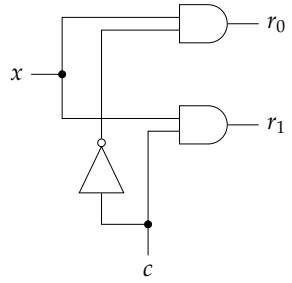
is described by the truth table

DEMUX2			
c	x	r_1	r_0
0	0	**?**	0
0	1	**?**	1
1	0	0	**?**
1	1	1	**?**

which can be used to derive the following implementation:

$$r_0 = \neg c \wedge x$$
$$r_1 = c \wedge x$$

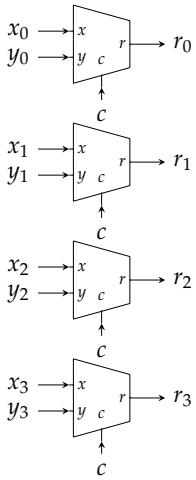
Circuit



Notes:

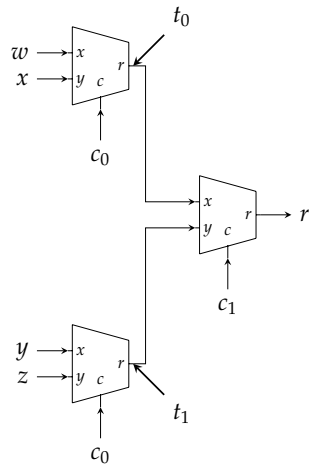
An Aside: Applying our design patterns

Circuit (2-input, 4-bit multiplexer via isolated replication)



Notes:

Circuit (4-input, 1-bit multiplexer via cascaded replication)



Notes:

"Building Block" Components (7)

Addition

- Just two building blocks can support most forms of arithmetic:
  1. a **half-adder**
     - has 2 inputs: x and y,
     - computes the 2-bit result $x + y$,
     - has 2 outputs: a sum s, and a carry-out co (which are the LSB and MSB of result),

     while
  2. a **full-adder**
     - has 3 inputs: x and y plus a carry-in ci,
     - computes the 2-bit result $x + y + ci$,
     - has 2 outputs: a sum s, and a carry-out co (which are the LSB and MSB of result),

  where all inputs and outputs are 1-bit.

Notes:

"Building Block" Components (8)

Addition

Definition

The behaviour of a **half-adder** component

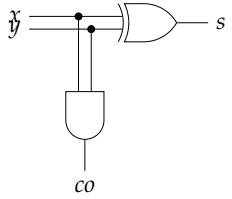
is described by the truth table

Half-Adder			
x	y	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

which can be used to derive the following implementation:

$$\begin{aligned} co &= x \wedge y \\ s &= x \oplus y \end{aligned}$$

Circuit



Notes:

"Building Block" Components (9)

Addition

Definition

The behaviour of a **full-adder** component

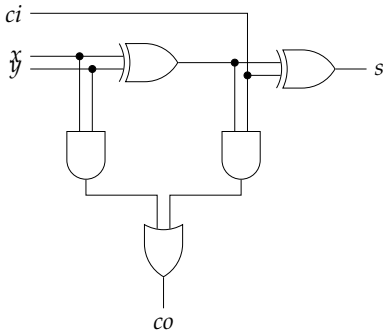
is described by the truth table

Full-Adder				
ci	x	y	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

which can be used to derive the following implementation:

$$\begin{aligned} co &= (x \wedge y) \vee (x \wedge ci) \vee (y \wedge ci) \\ &= (x \wedge y) \vee ((x \oplus y) \wedge ci) \\ s &= x \oplus y \oplus ci \end{aligned}$$

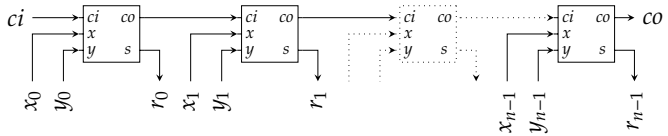
Circuit



Notes:

Notes:

Circuit (n-bit addition)



Notes:

- Just two building blocks can support most forms of comparison:
  1. an **equality comparator**
     - has 2 inputs x and y,
     - computes the 1 output as

$$r = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

  while
  2. a **less-than comparator**
     - has 2 inputs x and y,
     - computes the 1 output as

$$r = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{otherwise} \end{cases}$$

  where all inputs and outputs are 1-bit.

"Building Block" Components (12)

Comparison

Definition

The behaviour of an **equality comparator** component

is described by the truth table

Equal		
x	y	r
0	0	1
0	1	0
1	0	0
1	1	1

which can be used to derive the following implementation:

$$r = \neg(x \oplus y)$$

Circuit

Notes:

"Building Block" Components (13)

Comparison

Definition

The behaviour of a **less-than comparator** component

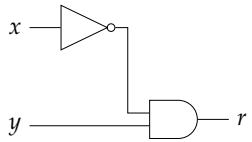
is described by the truth table

Less-Than		
x	y	r
0	0	0
0	1	1
1	0	0
1	1	0

which can be used to derive the following implementation:

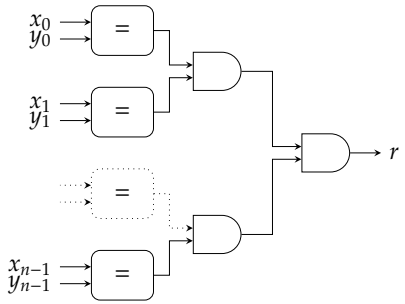
$$r = \neg x \wedge y$$

Circuit



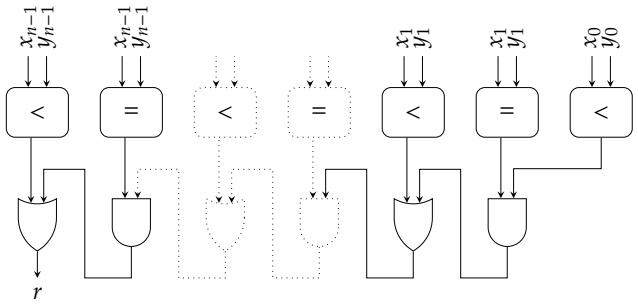
Notes:

Circuit (n-bit equality comparison)



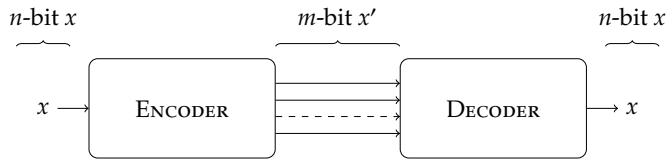
Notes:

Circuit (n-bit less-than comparison)



Notes:

- Informally at least, **encoder** and **decoder** components can be viewed as *translators*



or, more formally,

1. an n-to-m encoder translates an n-bit input into some m-bit code word, and
2. an m-to-n decoder translates an m-bit code word back into the same n-bit output

where if only one output (resp. input) is allowed to be 1 at a time, we call it a **one-of-many** encoder (resp. decoder).

Notes:

"Building Block" Components (16)
Control

- A *general* building block is impossible since it depends on the scheme for encoding/decoding: consider an example st.

1. to encode, take n inputs, say x_i for $0 \leq i < n$, and produce a unsigned integer x' that determines which $x_i = 1$,
2. to decode, take x' and set the right $x_i = 1$

where for all $j \neq i$, $x'_j = 0$ (so both are one-of-many).

Notes:

Definition (example encoder)

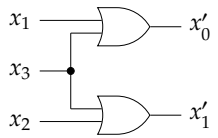
The example encoder is described by the truth table

Enc-4-to-2					
x_3	x_2	x_1	x_0	x'_1	x'_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

which can be used to derive the following implementation:

$$\begin{aligned} x'_0 &= x_1 \vee x_3 \\ x'_1 &= x_2 \vee x_3 \end{aligned}$$

Circuit (example encoder)



Notes:

Definition (example decoder)

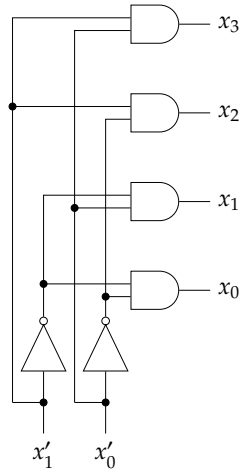
The example decoder is described by the truth table

Dec-2-to-4					
x'_1	x'_0	x_3	x_2	x_1	x_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

which can be used to derive the following implementation:

$$\begin{aligned} x_0 &= \neg x'_0 \wedge \neg x'_1 \\ x_1 &= x'_0 \wedge \neg x'_1 \\ x_2 &= \neg x'_0 \wedge x'_1 \\ x_3 &= x'_0 \wedge x'_1 \end{aligned}$$

Circuit (example decoder)



Notes:

- Problem: if we break the rules and set both $x_1 = 1$ *and* $x_2 = 1$, the encoder fails by producing

$$\begin{array}{rcccl} x'_0 & = & x_1 \vee x_3 & = & 1 \\ x'_1 & = & x_2 \vee x_3 & = & 1 \end{array}$$

  as the result.
- Solution: consider a **priority** encoder, where one input is given priority (or preference) over another.

Notes:

"Building Block" Components (19)
Control

Example

Imagine we want to give x_j priority over each x_k for $j > k$, so x_2 over x_1 and x_0 for example:

PriorityEnc-4-to-2					
x_3	x_2	x_1	x_0	x'_1	x'_0
0	0	0	1	0	0
0	0	1	**?**	0	1
0	1	**?**	**?**	1	0
1	**?**	**?**	**?**	1	1

Now, although potentially $x_0 = 1$ or $x_1 = 1$ the output gives priority to x_2: as long as $x_2 = 1$ and $x_3 = 0$, the output will be $x'_0 = 0$ and $x'_1 = 1$ irrespective of x_0 and x_1.

Notes:

© Daniel Page (cdsp@cs.bris.ac.uk)
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Conclusions

- Take away points:
  1. There are a *huge* number of challenges, even with (relatively) simple problems, e.g.,
     - how do we describe what the design should do?
     - how do we structure the design?
     - what sort of standard cell library do we use?
     - do we aim for the fewest gates?
     - do we aim for shortest critical path?
     - how do we cope with propagation delay and fan-out?
     - ...
  2. The three themes we've covered, i.e.,
     - high-level design patterns,
     - low-level, mechanical derivation and optimisation of Boolean expressions,
     - building-block components,

     allows us to address such challenges: in combination, they support development of effective (combinatorial) design and implementation.
  3. In many cases, use of appropriate **Electronic Design Automation (EDA)** tools can provide (semi-)automatic solutions.

Notes:

Additional Reading

- *Wikipedia: Combinational logic*. URL: http://en.wikipedia.org/wiki/Combinational_logic.
- D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009.
- W. Stallings. "Chapter 11: Digital logic". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013.
- A.S. Tanenbaum and T. Austin. "Section 3.2.2: Combinatorial circuits". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012.

Notes:

References

[1] *Wikipedia: Combinational logic*. URL: http://en.wikipedia.org/wiki/Combinational_logic (see p. 51).

[2] D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 51).

[3] W. Stallings. "Chapter 11: Digital logic". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013 (see p. 51).

[4] A.S. Tanenbaum and T. Austin. "Section 3.2.2: Combinatorial circuits". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 51).

Notes: