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:

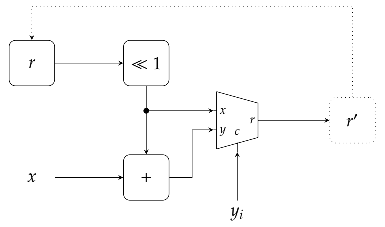
- Recap: our goal is to implement a bit-serial multiplier, i.e.,

Algorithm

Input: Two unsigned, n-bit, base-2 integers x and y
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
r ← 0
for i = n − 1 downto 0 step −1 do
    r ← 2 · r
    if y_i = 1 then
        r ← r + x
    end
end
return r
```

Circuit



as a case-study of data- and control-paths; we more or less have the data-path, but what about the control-path ...

Notes:

Question

Design an FSM-based component that replicates the behaviour of a loop counter, for example `i` within a C-style `for` loop such as

```
for( int i = m; i < n; i++ ) {
  ...
}
```

noting that

1. we'll look at *a* solution, not *all* solutions,
2. we'll need a mechanism that informs us when this is, plus also allows us to start iteration,
3. we'll allow the loop counter `i` to equal `n` once the loop is complete (as is the case in C), and
4. we'll consider 4-bit values of `i`, `m` and `n` st. this design is basically an "upgrade" to the previous, uncontrolled counter

and the `n` (the loop bound) here **isn't** necessarily n (the size of x and y) from the multiplier algorithm!

Notes:

A loop counter (1)
A control protocol

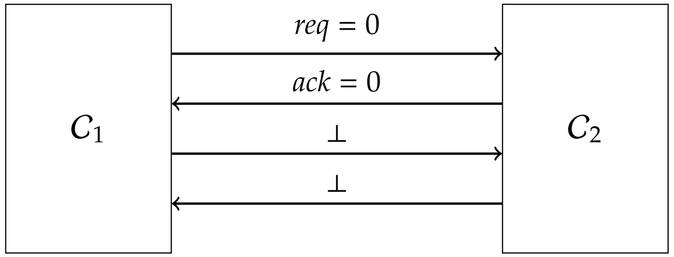
- Question: given a user C_1 of some component C_2, how does
  1. C_2 know when to start computation (e.g., when any input x is available), and
  2. C_1 know when computation has finished (e.g., when any output $r = f(x)$ is available).
- Solution(s):
  1. use a shared clock signal to synchronise events somehow, or
  2. use a simple **control protocol** based on two signals
     - 2.1 *req* (or **request**), and
     - 2.2 *ack* (or **acknowledge**).

Notes:

- A shared clock might *seem* the ideal option, but there are (at least) two scenarios where it doesn't work out so well:
  1. when C_2 and C_1 are physically separate, since synchronised clock distribution is hard(er), and
  2. when the number of steps C_2 takes is variable, since even if C_2 and C_1 are synchronised the latter still cannot tell when the former has completed a given computation.

A loop counter (2)
A control protocol

Algorithm



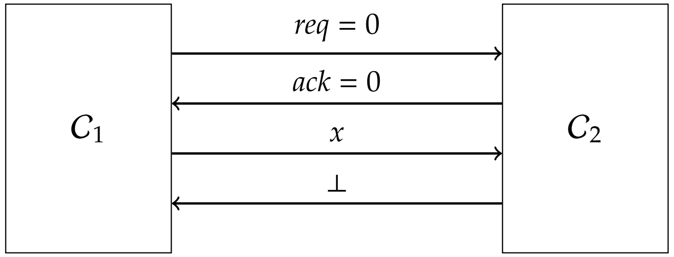
Notes:

-
  - Initially, $req = 0$ and $ack = 0$.
  - When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  - C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  - C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  - C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  - C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  - Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



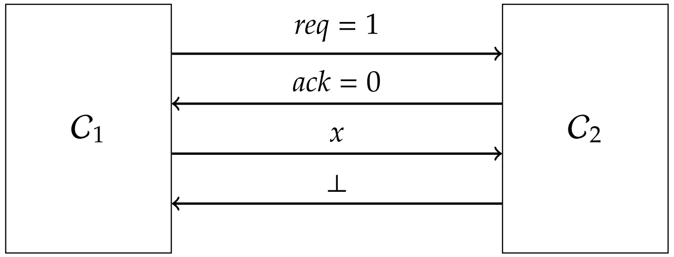
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



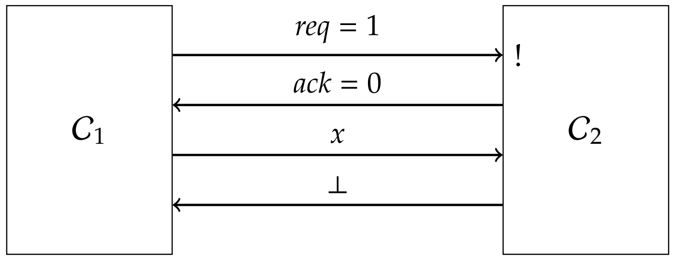
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm

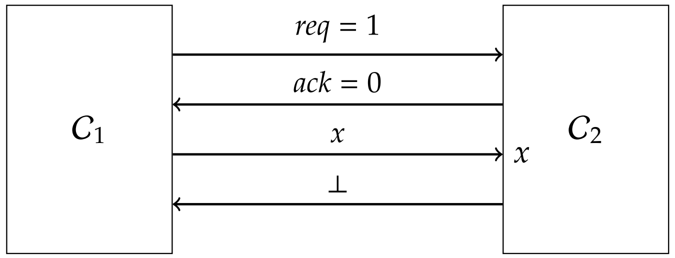


Notes:

- – Initially, *req* = 0 and *ack* = 0.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets *req* = 1 to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets *ack* = 1.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets *req* = 0.
  – C_2 notices this, and concludes that the computation is finished: it sets *ack* = 0 then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both *req* = 0 and *ack* = 0, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)
A control protocol

Algorithm



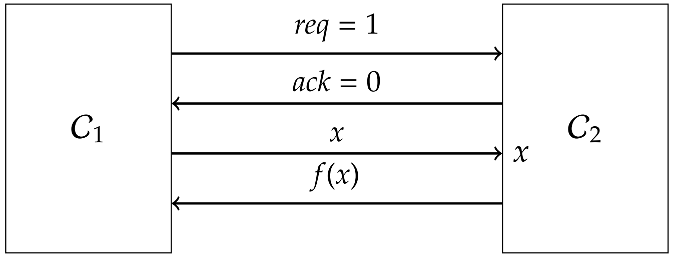
Notes:

- – Initially, *req* = 0 and *ack* = 0.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets *req* = 1 to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets *ack* = 1.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets *req* = 0.
  – C_2 notices this, and concludes that the computation is finished: it sets *ack* = 0 then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both *req* = 0 and *ack* = 0, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



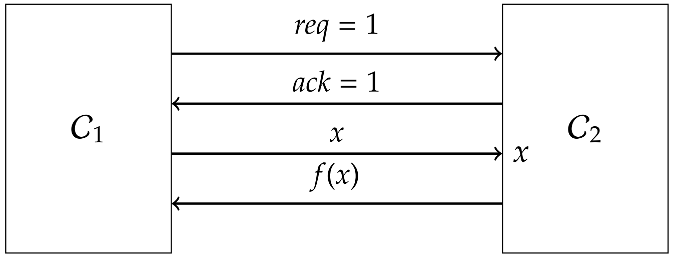
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



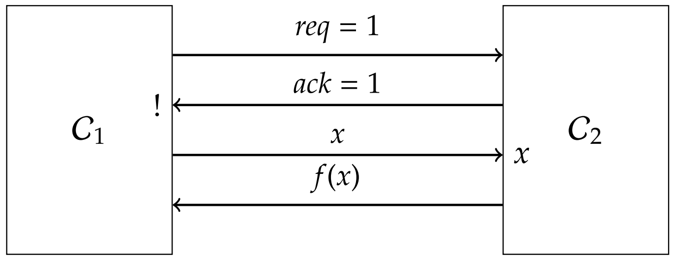
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



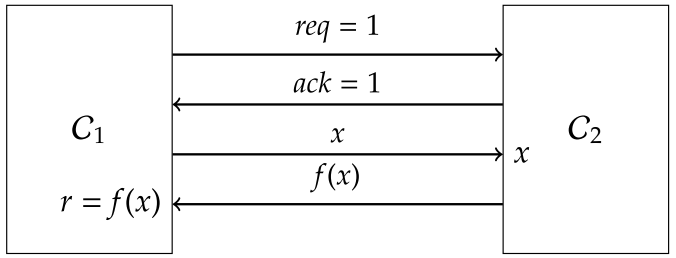
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



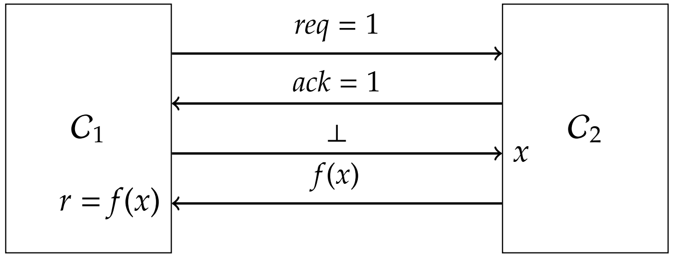
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



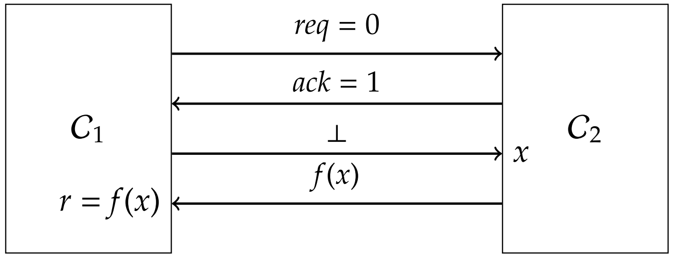
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm

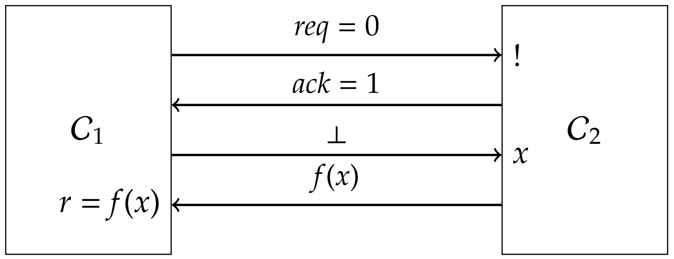


Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)
A control protocol

Algorithm

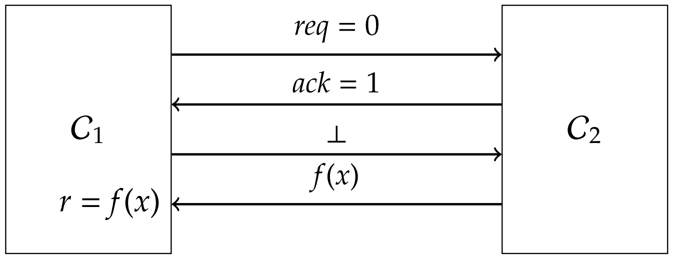


Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)
A control protocol

Algorithm



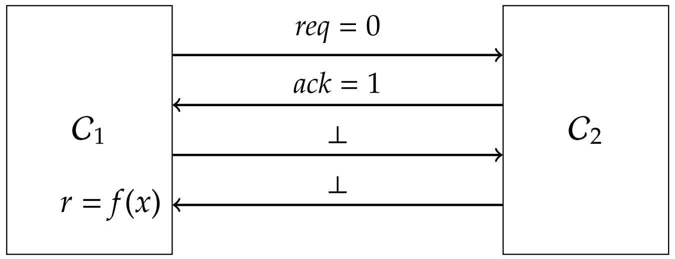
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm



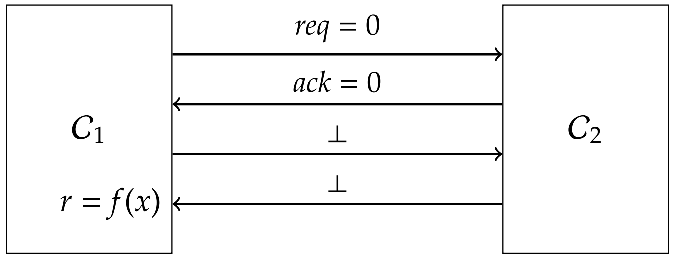
Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)

A control protocol

Algorithm

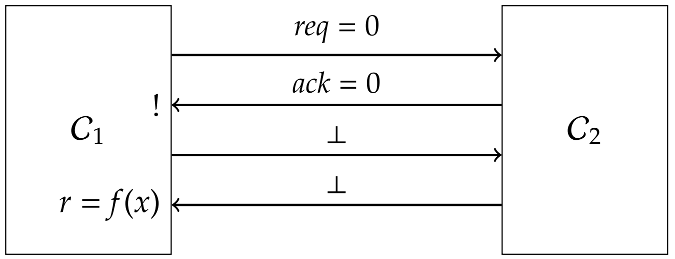


Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)
A control protocol

Algorithm

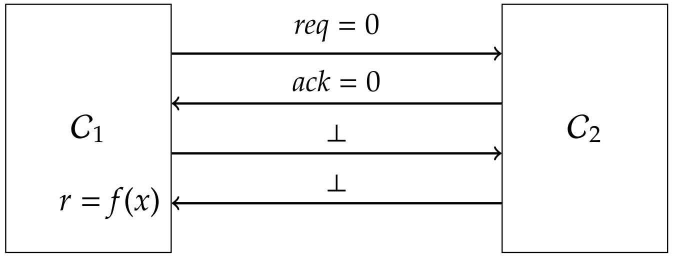


Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (2)
A control protocol

Algorithm

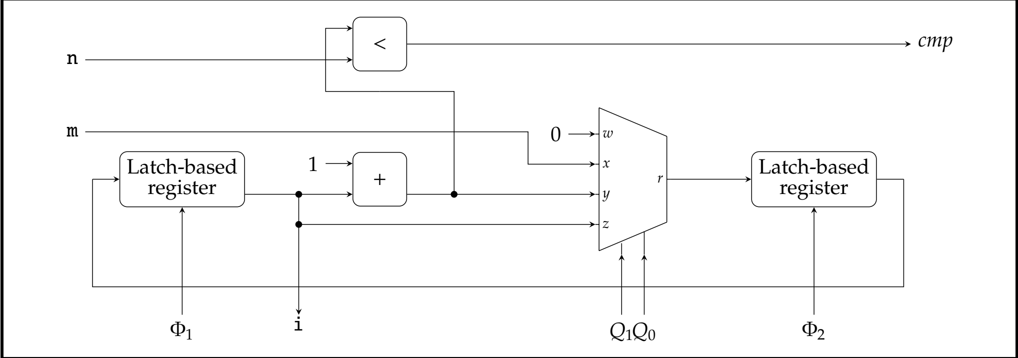


Notes:

- – Initially, $req = 0$ and $ack = 0$.
  – When C_1 wants to compute something, it first drives any input values (say x) then sets $req = 1$ to signal they are ready for use.
  – C_2 notices this, and starts computation. Eventually the computation is complete, meaning $r = f(x)$; to signal this to C_1, it sets $ack = 1$.
  – C_1 notices this, and concludes that the computation is finished: it latches r, and finally sets $req = 0$.
  – C_2 notices this, and concludes that the computation is finished: it sets $ack = 0$ then waits for the next request.
  – C_1 notices this, and proceeds to use r for whatever purpose it was computed.
  – Since both $req = 0$ and $ack = 0$, the process can now begin again from the start whenever C_1 needs to perform another computation.

A loop counter (3)
A data-path

Circuit (data-path)



Notes:

- The general structure here is the same as the previous, uncontrolled counter example; the only difference is the the diagram is read from left-to-right rather than bottom-to-top. Otherwise, there is still

  1. an input register (a collection of D-type latches) enabled by Φ_1,
  2. some combinatorial logic that computes the next value of the counter from the current value, and
  3. an output register (a collection of D-type latches), enabled by Φ_2.

  When $\Phi_2 = 1$ the output register is updated with whatever value is computed by the combinatorial logic: this is dictated by the Q input, acting as the multiplexer control signal. When $\Phi_1 = 1$, whatever was stored in the output register is fed back around and used to update the input register; after this, the cycle repeats.

A loop counter (4)
A control-path

- Idea: we can use an FSM to implement the control protocol.
  - The FSM can be in one of 4 states, namely
    - in S_{wait} it waits for a request (i.e., for $req = 1$),
    - in S_{init} it uses any input to initialise itself (e.g., setting the initial loop counter value),
    - in S_{step} it performs an iteration of the loop, and
    - in S_{done} it waits for $req = 0$ (while setting $ack = 1$) once the loop is complete.
  - Since $2^2 = 4$, we can represent them using 4 concrete values, namely

$$\begin{array}{lclcl}
S_{wait} & \mapsto & \langle 0,0\rangle & \equiv & 00_{(2)} \\
S_{init} & \mapsto & \langle 1,0\rangle & \equiv & 01_{(2)} \\
S_{step} & \mapsto & \langle 0,1\rangle & \equiv & 10_{(2)} \\
S_{done} & \mapsto & \langle 1,1\rangle & \equiv & 11_{(2)}
\end{array}$$

    and capture

    1. $Q = \langle Q_0, Q_1\rangle \equiv$ the current state
    2. $Q' = \langle Q'_0, Q'_1\rangle \equiv$ the next state

    in a 2-bit register (i.e., via 2 latches or flip-flops).

Notes:

A loop counter (5)

A control-path

Algorithm (control-path, tabular)

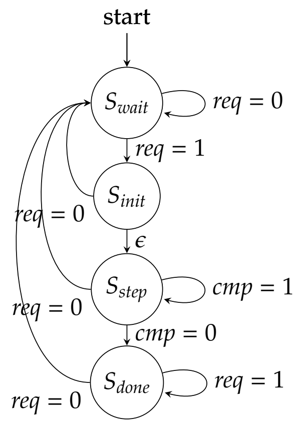
		δ		ω	
	Q	Q'		ack	
		$cmp = 0$	$cmp = 1$	$cmp = 0$	$cmp = 1$
$req = 0$	S_{wait}	S_{wait}	S_{wait}	0	0
	S_{init}	S_{wait}	S_{wait}	0	0
	S_{step}	S_{wait}	S_{wait}	0	0
	S_{done}	S_{wait}	S_{wait}	1	1
$req = 1$	S_{wait}	S_{init}	S_{init}	0	0
	S_{init}	S_{step}	S_{step}	0	0
	S_{step}	S_{done}	S_{step}	0	0
	S_{done}	S_{done}	S_{done}	1	1

Notes:

A loop counter (6)

A control-path

Algorithm (control-path, diagram)



Notes:

A loop counter (7)
A control-path

Algorithm (control-path, truth table)

Rewriting the abstract labels yields the following concrete truth table:

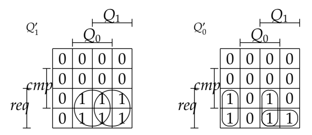
				δ		ω
req	*cmp*	Q_1	Q_0	Q'_1	Q'_0	*ack*
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	1
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	0	1	1	0	0
1	0	1	0	1	1	0
1	0	1	1	1	1	1
1	1	0	0	0	1	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	1	1	1

Notes:

A loop counter (8)
A control-path

Circuit (control-path, δ)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions:

$$\begin{array}{lllllllll}
Q'_1 = (& req & & & & & \wedge & Q_0 &) \vee \\
(& req & & & \wedge & Q_1 & & &) \\
\\
Q'_0 = (& req & & & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \vee \\
(& req & \wedge & & \wedge & Q_1 & \wedge & Q_0 &) \vee \\
(& req & \wedge & \neg cmp & \wedge & Q_1 & & &)
\end{array}$$

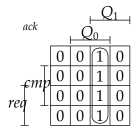
Notes:

A loop counter (9)

A control-path

Circuit (control-path, ω)

Translating the truth table into a set of Karnaugh maps



yields the following Boolean expressions:

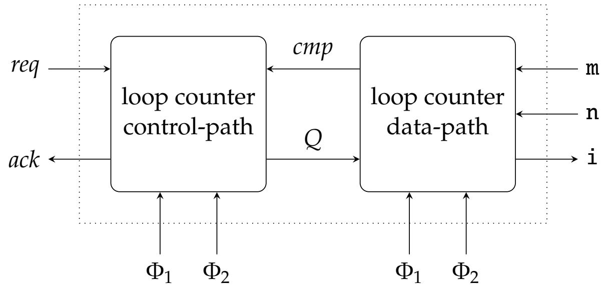
$$ack = Q_1 \wedge Q_0$$

Notes:

Conclusions

- Next steps (or, the lab. session):
  1. We now have the loop counter implemented as specified, i.e.,

Circuit (data- *and* control-paths)



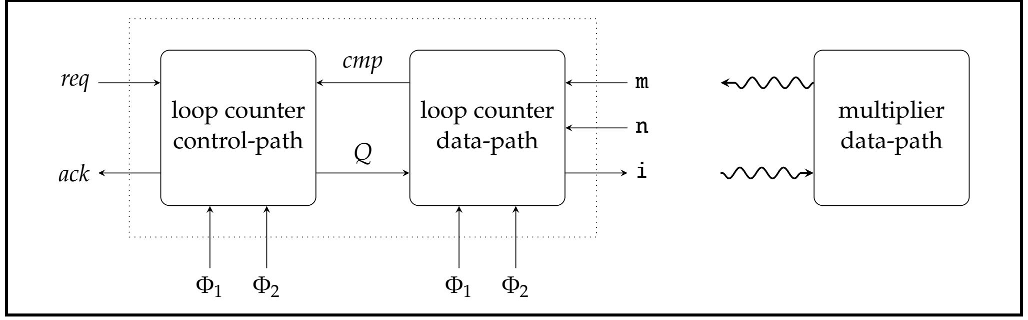
Notes:

- Now we understand how the FSM operates as a control path, we can be more specific about how the data path is controlled. Specifically, it should now be clear that
  1. if $Q = \langle 0, 0 \rangle \mapsto S_{wait}$, then the multiplexer updates the output register with 0,
  2. if $Q = \langle 1, 0 \rangle \mapsto S_{init}$ then the multiplexer updates the output register with `m`, i.e., the initial counter value,
  3. if $Q = \langle 0, 1 \rangle \mapsto S_{step}$ then the multiplexer updates the output register with `i` + 1, i.e., the incremented counter value produced by the adder,
  4. if $Q = \langle 1, 1 \rangle \mapsto S_{done}$ then the multiplexer updates the output register with `i`, i.e., the current counter value.

Conclusions

- Next steps (or, the lab. session):
  1. We now have the loop counter implemented as specified, i.e.,

Circuit (data- *and* control-paths, *plus* multiplier data-path)



  2. The next challenge is then *using* it to realise the original goal, e.g., specifying
     - any additional data-path components required, and
     - how loop counter (the control-path) controls them

     so we end up with a bit-serial multiplier.

Notes:

- Now we understand how the FSM operates as a control path, we can be more specific about how the data path is controlled. Specifically, it should now be clear that
  1. if $Q = \langle 0, 0 \rangle \mapsto S_{wait}$, then the multiplexer updates the output register with 0,
  2. if $Q = \langle 1, 0 \rangle \mapsto S_{init}$ then the multiplexer updates the output register with `m`, i.e., the initial counter value,
  3. if $Q = \langle 0, 1 \rangle \mapsto S_{step}$ then the multiplexer updates the output register with `i` + 1, i.e., the incremented counter value produced by the adder,
  4. if $Q = \langle 1, 1 \rangle \mapsto S_{done}$ then the multiplexer updates the output register with `i`, i.e., the current counter value.

Additional Reading

- *Wikipedia: Computer Arithmetic*. URL: `http://en.wikipedia.org/wiki/Category:Computer_arithmetic`.
- D. Page. "Chapter 7: Arithmetic and logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009.
- B. Parhami. "Part 3: Multiplication". In: *Computer Arithmetic: Algorithms and Hardware Designs*. 1st ed. Oxford University Press, 2000.
- W. Stallings. "Chapter 10: Computer arithmetic". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013.
- A.S. Tanenbaum and T. Austin. "Section 3.2.2: Arithmetic circuits". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012.

Notes:

References

[1] *Wikipedia: Computer Arithmetic.* URL: http://en.wikipedia.org/wiki/Category:Computer_arithmetic (see p. 63).

[2] D. Page. "Chapter 7: Arithmetic and logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 63).

[3] B. Parhami. "Part 3: Multiplication". In: *Computer Arithmetic: Algorithms and Hardware Designs*. 1st ed. Oxford University Press, 2000 (see p. 63).

[4] W. Stallings. "Chapter 10: Computer arithmetic". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013 (see p. 63).

[5] A.S. Tanenbaum and T. Austin. "Section 3.2.2: Arithmetic circuits". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 63).

Notes: