Specification and Verification II

- **Topic of course is the** *Specification and Verification of Hardware*
- **Assumes familiarity with** *Specification and Verification I*
  (which concerns software, particularly using Hoare logic)
- **The two courses are really a single course**

The notes contain general and background material for the course.
Some of the material in them may not be covered in the lectures.
Some details and examples are only presented in the lectures.
The examinable material is what is actually covered in the lectures

---

Starting today

- **Hardware oriented Hoare logic examples**
  - **apply** *Specification and Verification I* **ideas to hardware**
- **Modelling data**
  - **words as numbers or as bit arrays**
- **Programs as hardware**
  - **synthesis to state machines**
- **Compare program behaviour with hardware behaviour**
  - **intermediate states visible**
- **Motivate temporal logic**
  - **need to specify more than relationship between input and final result**

Hoare Logic, Higher Order Logic and Temporal Logic

- **Hoare logic can be used to verify programs in HDLs**
- **Hoare logic can be embedded in higher order logic**
  - **see last part of** *Specification and Verification I*
- **Higher order logic will be used to represent hardware structures**
- **Temporal logic (see later):**
  - **is used to specify properties**
  - **can be embedded in higher order logic**
- **Hoare Logic is for data reasoning, temporal logic for time (control)**
- **Need to chose appropriate logic – all live inside higher order logic**
- **Goal: software and hardware modelled in same language**
  - **programming languages get hardware features: .... SystemC**
  - **hardware description languages get programming features: .. SystemVerilog**

Hardware Oriented Programs

- **Hoare logic can be used to verify hardware algorithms**
  - can reason about programs to develop hardware
  - not yet 'Industry Standard' practice
  - interesting research direction: applications to hardware/software co-design?

- **Hoare logic *ideas* appear in some industrial methods**
  - Intel's Symbolic Trajectory Evaluation (STE)

    `{stimulus} <hardware model> {response}`
  - Assertion Based Verification (ABV) for hardware

    annotates HDL source with assertions

Hoare Logic applied to hardware algorithms

- **Examples: addition and multiplication**
- **Initially natural numbers will represent words**
  - leads to messy details
  - later a type of words is introduced
- **We will multiply natural numbers a and b**
  - assume they can be represented with n bits
- **Write ab to abbreviate $a \times b$**
- **a_i is i-th bit of the binary representation of a (a_0 being the least significant bit)**

$$a = 2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + \cdots + 2^0 a_0$$

hence

$$\begin{aligned} ab &= (2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + \cdots + 2^0 a_0)b \\ &= 2^{n-1}a_{n-1}b + 2^{n-2}a_{n-2}b + \cdots + 2^0 a_0 b \\ &= a_{n-1}2^{n-1}b + a_{n-2}2^{n-2}b + \cdots + a_0 2^0 b \end{aligned}$$

Binary multiplication algorithm

- **Multiplying by 2 corresponds to:**
  - **shifting one place to the left**
  - **adding a 0 as the least significant bit**
- **Denote this operation by $b \mapsto b^\frown 0$, then:**

$$\begin{aligned}
2^0 b &= b \\
2^1 b &= b^\frown 0 \\
2^2 b &= b^\frown 00 \\
&\vdots \\
2^n b &= b^\frown \underbrace{0 \cdots 0}_{n\ 0s}
\end{aligned}$$

- **Recall:** $ab = a_{n-1}2^{n-1}b + a_{n-2}2^{n-2}b + \cdots + a_0 2^0 b$
- **Thus product of a and b is given by the sum:**

$$\begin{array}{lr}
 & a_0 b \\
+ & a_1 b^\frown 0 \\
+ & a_2 b^\frown 00 \\
+ & a_3 b^\frown 000 \\
 & \vdots \\
+ & a_{n-1} b^\frown 0 \cdots 0
\end{array}$$

  - **the i^{th} row is either all zeros (if a_i is 0)**
  - **or b shifted i places to the left (if a_i is 1)**
- **a, b need n-bits $\Rightarrow$ product needs $2n$ bits**

Extracting bits and subwords

- **Let** $\texttt{A}[n]$ **denote the** n**-th bit of the binary representation of** $\texttt{A}$

- $\texttt{A}[n]$ **is a number** 1 **or** 0

- $\texttt{A}[0]$ **is the least significant bit**

- **Thus:**

$$\texttt{A}[n] = (\texttt{A div } 2^n) \texttt{ mod } 2$$

- **Define** $\texttt{A}[m:n]$ **to be the** **numerical value** **of the word comprising bits** n **upto to** m **of** $\texttt{A}$**:**

$$\begin{cases} 2^{m-n}\texttt{A}[m] + 2^{m-n-1}\texttt{A}[m{-}1] + \ \cdots \ + 2^0\texttt{A}[n] & \textbf{if } m > n \\ \texttt{A}[n] & \textbf{if } m = n \\ 0 & \textbf{if } m < n \end{cases}$$

- **Later we'll represent words as bit-strings instead of as numbers**

Hoare logic verification of a multiplier

- **Add-shift multiplication program:**

```
I := 0; PROD := 0;
WHILE I < N DO
 BEGIN PROD := PROD + A[I] × (2^I× B);
       I := I + 1;
 END
```

- **Annotated Hoare specification:**

$\{\mathtt{A} = a \ \wedge\ \mathtt{B} = b \ \wedge\ a < 2^{\mathtt{N}} \ \wedge\ b < 2^{\mathtt{N}} \ \wedge\ \mathtt{N} > 0\}$

```
I := 0; PROD := 0;
WHILE I < N DO
```
$\{\mathtt{I} \leq \mathtt{N} \ \wedge\ 2^{\mathtt{I}}\mathtt{A[N-1:I]B} + \mathtt{PROD} = ab\}$
```
 BEGIN PROD := PROD + A[I] × (2^I× B);
       I := I + 1;
 END
```
$\{\mathtt{PROD} = a \times b\}$

- **Routine (not trivial) to verify using Hoare Logic**
  - **reasoning about `div` and `mod` is** *horrible*

Using FOR-commands instead of WHILE

$\vdash \{\mathtt{A} = a \;\wedge\; \mathtt{B} = b \;\wedge\; a < 2^{\mathtt{N}} \;\wedge\; b < 2^{\mathtt{N}} \;\wedge\; \mathtt{N} > 0\}$

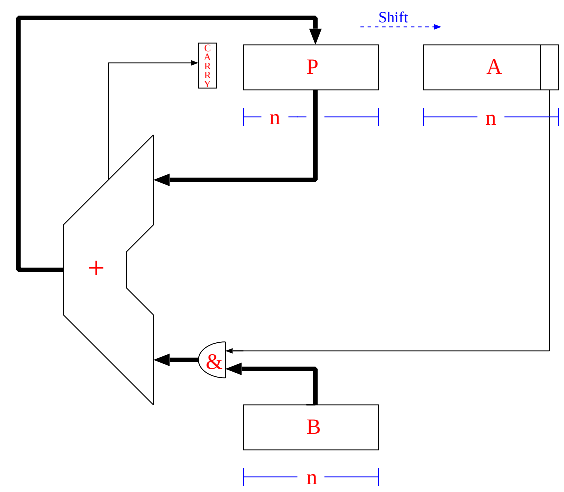
```
PROD := 0;
FOR I := 0 UNTIL N-1 DO
 BEGIN PROD := PROD + A[I]×B;
       B := 2×B;
 END
```

$\{\mathtt{PROD} = a \times b\}$

- **Program corresponds directly to hardware (i.e. more like HDL)**
  - **three registers** `A`, `B` **and** `PROD`
  - **initially** `PROD` **is set to** 0
  - `A` **and** `B` **contain numbers to be multiplied**
- `I`**-th step of the multiplication:**
  - **adding** `A[I]` $\times$ `B` **to** `PROD`
  - **then shifting** `B` **one bit to the left (i.e. multiplying it by** 2**)**

Textbook multiplier

- **Simple textbook add-shift multiplier:**



- **Optimised version of naive algorithm**

- **Can apply Hoare logic methods to verify correctness**
  - see notes for (horrible) details of Hoare-style proof

Words as bit-strings (see notes for full details)

- **Distinguish words from numbers – different type**
  - **Advantages: corresponds more to intuition – words have a size**
  - **Disadvantage: can't use off-the-shelf theory of arithmetic**
- **Size of a word is denoted by** $|w|$
- $n^{\mathbf{th}}$ **bit of** w **denoted by** $w\texttt{[}n\texttt{]}$
- $w\texttt{[}m:n\texttt{]}$ **denotes bits** m **to** n **of** w
- **The word corresponding to a bit** b **is** $\texttt{Bw}(b)$
- $\texttt{Bv}(b)$ **is the number represented by bit** b
- $\texttt{V}(w)$ **is the natural number represented by word** w
- $\texttt{W}\ n$ **maps number** m **to the** n**-bit word representing it**
- **Concatenation of** w_1 **with** w_2 **denoted by** $w_1 {}^\frown w_2$
- $w\{n{\leftarrow}b\}$ **denotes a word such that** $w\texttt{[}n\texttt{]} = b$ **and is identical to** w **at all other bit positions (pad** w **with** 0**s at left if** $n \geq |w|$**)**
- **The addition** $w_1 \uplus w_2$ **of** w_1 **and** w_2 **is defined by:**
  $w_1 \uplus w_2 \ = \ \texttt{W}\ (\texttt{max}(|w_1|, |w_2|) + 1)\ (\texttt{V}(w_1) + \texttt{V}(w_2))$
- $b.w$ **equals** w **if** $b = \texttt{T}$ **and equals** $\texttt{W}\ |w|\ 0$ **if** $b = \texttt{F}$

Words vs bits

- $w\texttt{[}n:n\texttt{]}$ **is the** 1**-bit word consisting of** $w\texttt{[}n\texttt{]}$
- $w\texttt{[}n\texttt{]} : \mathit{bool}$
- $w\texttt{[}n:n\texttt{]} : \mathit{word}$
- **Bits and** 1**-bit words are different types**
- **The word corresponding to a bit** b **is** $\texttt{Bw}(b)$
- **Thus:** $\texttt{Bw}(b)\texttt{[}0\texttt{]} \;=\; b$

Representing Numbers

- **Natural number:** $b_{n-1} \cdots b_0$ **represents**

$$2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$$

- **Integer:** $b_{n-1} \cdots b_0$ **represents**

$$-2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$$

  - **this is the two's complement representation**

- $\mathtt{V}(w)$ **is the natural number represented by a** w

$$\mathtt{V}(b_{n-1} \cdots b_0) \;=\; 2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$$

- **Words can represent other values**
  - **e.g. floating point numbers; opcodes**

- $\mathtt{Bv}(b)$ **is the number represented by** b

$$\mathtt{Bv}(\mathtt{T}) \;=\; 1 \quad \textbf{and} \quad \mathtt{Bv}(\mathtt{F}) \;=\; 0$$

Arithmetic on bits and words

- **The sum of bits a and b and a carry-in bit c**
  - **is computed by $a \oplus b \oplus c$ (where $\oplus$ is 'exclusive or')**
  - **and the carry-out by $(a \wedge b) \vee (c \wedge (a \oplus b))$**
- **This is verified by:**

$$
\begin{aligned}
\mathtt{Bv}(a \oplus b \oplus c) &= (\mathtt{Bv}(a) + \mathtt{Bv}(b) + \mathtt{Bv}(c))\ \mathtt{mod}\ 2 \\
\mathtt{Bv}((a \wedge b) \vee (c \wedge (a \oplus b))) &= (\mathtt{Bv}(a) + \mathtt{Bv}(b) + \mathtt{Bv}(c))\ \mathtt{div}\ 2
\end{aligned}
$$

Verification by enumeration

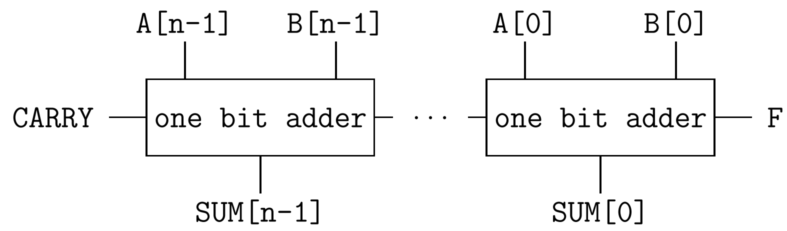
- **Sum:**

a	b	c	$\texttt{Bv}(a \oplus b \oplus c)$	$(\texttt{Bv}(a) + \texttt{Bv}(b) + \texttt{Bv}(c))$ $\texttt{mod}$ 2
1	1	1	1	1
1	1	0	0	0
1	0	1	0	0
1	0	0	1	1
0	1	1	0	0
0	1	0	1	1
0	0	1	1	1
0	0	0	0	0

- **Carry:**

a	b	c	$\texttt{Bv}((a \wedge b) \vee (c \wedge (a \oplus b)))$	$(\texttt{Bv}(a) + \texttt{Bv}(b) + \texttt{Bv}(c))$ $\texttt{div}$ 2
1	1	1	1	1
1	1	0	1	1
1	0	1	1	1
1	0	0	0	0
0	1	1	1	1
0	1	0	0	0
0	0	1	0	0
0	0	0	0	0

Verification of a ripple-carry adder of any size



- **Let R be:**

  $2^{\mathtt{I}}\mathtt{Bv}(\mathtt{CARRY}) + \mathtt{V}(\mathtt{SUM[I-1:0]}) = \mathtt{V}(\mathtt{A[I-1:0]}) + \mathtt{V}(\mathtt{B[I-1:0]}) \ \wedge$
  $\mathtt{A} = w_1 \ \wedge \ \mathtt{B} = w_2$

- **Consider the following annotated specification:**

  $\{\mathtt{A} = w_1 \ \wedge \ \mathtt{B} = w_2 \ \wedge \ \mathtt{SUM} = \mathtt{W\ N}\ 0 \ \wedge \ \mathtt{CARRY} = \mathtt{F} \ \wedge$
  $|w_1| \leq \mathtt{N} \ \wedge \ |w_2| \leq \mathtt{N} \ \wedge \ \mathtt{N} > 0\}$

  ```
  FOR I := 0 UNTIL N-1 DO {R}
   BEGIN
    SUM[I]:=A[I]⊕B[I]⊕CARRY;
    CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
   END
  ```

  $\{2^{\mathtt{N}}\mathtt{Bv}(\mathtt{CARRY}) + \mathtt{V}(\mathtt{SUM[N-1:0]}) \ = \ \mathtt{V}(\mathtt{A[N-1:0]}) + \mathtt{V}(\mathtt{B[N-1:0]})$
  $\mathtt{A} = w_1 \ \wedge \ \mathtt{B} = w_2\}$

- `A`, `B` **are** `N`**-bit words,** `SUM`, `CARRY` **are truthvalues,** `I` **is an integer**

- **Proof horrible (omitted)**

Word multiplication program

- **Simple add-shift multiplication**

- **Annotated correctness specification:**

$\{\mathtt{V}(\mathtt{A}) = a \ \wedge \ \mathtt{V}(\mathtt{B}) = b \ \wedge \ \mathtt{PROD} = \mathtt{W}(2\mathtt{N})0 \ \wedge$
$\ |\mathtt{A}| \leq \mathtt{N} \ \wedge \ |\mathtt{B}| \leq \mathtt{N} \ \wedge \ \mathtt{N} > 0\}$

```
 FOR I:=0 UNTIL N-1 DO
```

$\quad \{2^{\mathtt{I}}\mathtt{V}(\mathtt{A[N-1:I]})b + \mathtt{V}(\mathtt{PROD}) = ab \ \wedge \ \mathtt{V}(\mathtt{B}) = 2^{\mathtt{I}}b\}$

```
  BEGIN
   PROD := PROD ⊎ A[I].B;
   B := B⌢0
  END
```

$\{\mathtt{V}(\mathtt{PROD}) = ab\}$

- **Can generate VCs and prove them (horrible – omitted)**

Topic shift: From programs to hardware (i.e. synthesis)

- **Consider a ripple-carry adder**

```
FOR I := 0 UNTIL N-1 DO
 BEGIN
  SUM[I]:=A[I]⊕B[I]⊕CARRY;
  CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
 END
```

- **If a particular value of `N` is fixed, then the program can be unrolled into the normal circuit for an adder.**

- **For example take `N` $= 3$ to get:**

```
FOR I := 0 UNTIL 2 DO
 BEGIN
  SUM[I]:=A[I]⊕B[I]⊕CARRY;
  CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
 END
```

N=3 adder

- **3-bit adder:**

```
FOR I := 0 UNTIL 2 DO
 BEGIN
  SUM[I]:=A[I]⊕B[I]⊕CARRY;
  CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
 END
```

- **Assuming initially `CARRY` = `F`; `FOR`-command unrolls to:**

```
SUM[0]:=A[0]⊕B[0]⊕F;
CARRY:=(A[0]∧B[0])∨(F∧(A[0]⊕B[0]));
SUM[1]:=A[1]⊕B[1]⊕CARRY;
CARRY:=(A[1]∧B[1])∨(CARRY∧(A[1]⊕B[1]));
SUM[2]:=A[2]⊕B[2]⊕CARRY;
CARRY:=(A[2]∧B[2])∨(CARRY∧(A[2]⊕B[2]));
```

- **Symbolically executing yields logic equations:**

```
SUM[0]:=A[0]⊕B[0];
SUM[1]:=A[1]⊕B[1]⊕(A[0]∧B[0]);
SUM[2]:=A[2]⊕B[2]⊕
          ((A[1]∧B[1])∨((A[0]∧B[0])∧(A[1]⊕B[1])));
CARRY :=(A[2]∧B[2])∨
        (((A[1]∧B[1])∨((A[0]∧B[0])∧(A[1]⊕B[1])))∧
         (A[2]⊕B[2]));
```

Combinational logic

- **Derived program is combinational logic:**

```
SUM[0]:=A[0]⊕B[0];
SUM[1]:=A[1]⊕B[1]⊕(A[0]∧B[0]);
SUM[2]:=A[2]⊕B[2]⊕
          ((A[1]∧B[1])∨
           ((A[0]∧B[0])∧(A[1]⊕B[1])));
CARRY :=(A[2]∧B[2])
        ∨
        (((A[1]∧B[1])∨
          ((A[0]∧B[0])∧(A[1]⊕B[1])))∧
         (A[2]⊕B[2]));
```

- **These are independent assignments**
  - **boolean expressions for computing the values of** `SUM` **and** `CARRY` **directly in terms of the** `A[0]`, `A[1]`, `A[2]`, `B[0]`, `B[1]` **and** `B[2]`

- **This process yields logic for adders of arbitrary (fixed) bit-widths**

- **Hoare Logic verifies** ***any*** **adder generated this way**

What about non-combinational logic?

- **Unrolling commands to combinational logic is sensible for the adder**
- **Less so for multipliers**
  - **straightforward to unroll a multiplier into combinational logic**
  - **but resulting Boolean expressions will be huge**
  - **evaluating in one clock cycle likely to make the cycle time too slow**
- **Usually multipliers are sequential machines**
  - **compute the product over a number of cycles**
  - **might do the add and shift in a single cycle which would take `N` cycles**
  - **might do add and shift on separate cycles, taking `2N` shorter cycles**
- **Decision of whether to implement a particular function as combinational or sequential logic, and if sequential, how much to do each cycle, is a decision which depends on engineering issues**

Specifying cycles

- **Abstract view of multiplier:**
  - **computes a single state change**
  - **from initial values of the registers**
  - **to final values**
- **Adequate for functional correctness**
  - **i.e. it does multiplication**
- **Less abstract views needed for timing analysis**

HDLs and events

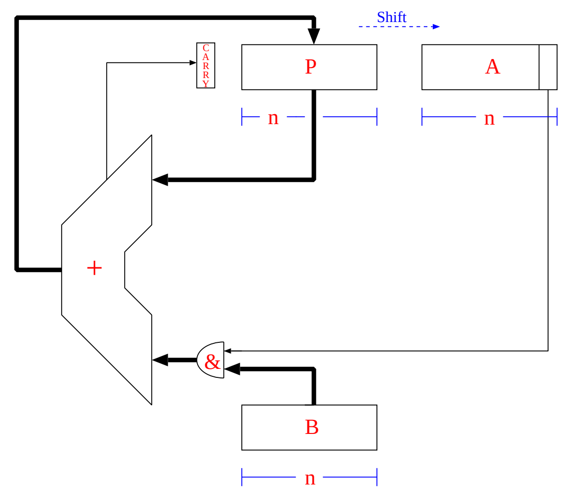
- **HDLs allow operations to be scheduled to clock cycles**

- **Statements can be prefixed by `@`**
  - **the symbol `@` introduces an** *event control*

- **Multiplier that takes `N` cycles:**

`R = CARRY⌢P⌢A`

```
FOR I := 0 UNTIL N-1 DO
 @R := (R[0].B ⊎ R[2N-1:N])⌢R[N-1:1]
```



- **Multiplier that takes `2N` cycles:**

```
FOR I := 0 UNTIL N-1 DO
 BEGIN
  @SUM := R[0].B ⊎ R[2N-1:N];
  @R := SUM⌢R[N-1:1]
 END
```

- **In Verilog, event controls can be more detailed**
  - `@(posedge clk)` **or** `@(negedge clk)`

Need more than Hoare Logic

- **Programs with added event controls can still be reasoned about using Floyd Hoare logic**
  - **relation between initial and final state unchanged**
  - `@`**'s just determine intermediate states at clock ticks**

- **Consider this silly program:**

```
FOR I := 0 UNTIL N-1 DO
 BEGIN
  @SUM := R[0].B ⊎ R[2N-1:N];
  B := ¬B;
  @R := SUM⁀R[N-1:1];
  B := ¬B;
 END
```

- **Same initial-final relation, but** `B` **oscillates**

- **Hoare specifications only deal with initial-final relation, not intermediate states**

- *Temporal logic* **enables properties of intermediate states to be specified**
  - **e.g.** `B` **stable (false for silly program above)**

Division program from Specification and Verification I

- **Division program:**

```
R:=X;
Q:=0;
WHILE Y≤R DO
  BEGIN R:=R-Y; Q:=Q+1 END
```

- **Implemented as a machine**
  - **registers `X`, `Y`, `Q` and `R`**
  - **a subtracter and incrementer**
  - **on each cycle: subtract `Y` from `R`; add `1` to `Q`**

- *Specification and Verification I:*
  - **program executes once and stops (maybe)**

- *Specification and Verification II*:
  - **program executes continuously**
  - **body of loop executed as combinational logic**

Our toy language becomes an HDL

- **To emphasize the continuously-running nature of hardware, recast division program as (where `FOREVER` is `WHILE T DO`):**

```
FOREVER
  IF Load=1
   THEN X:=In1; Y:=In2; DONE:=0; R:=X; Q:=0
   ELSE IF Y≤R THEN R:=R-Y; Q:=Q+1
               ELSE DONE:=1
```

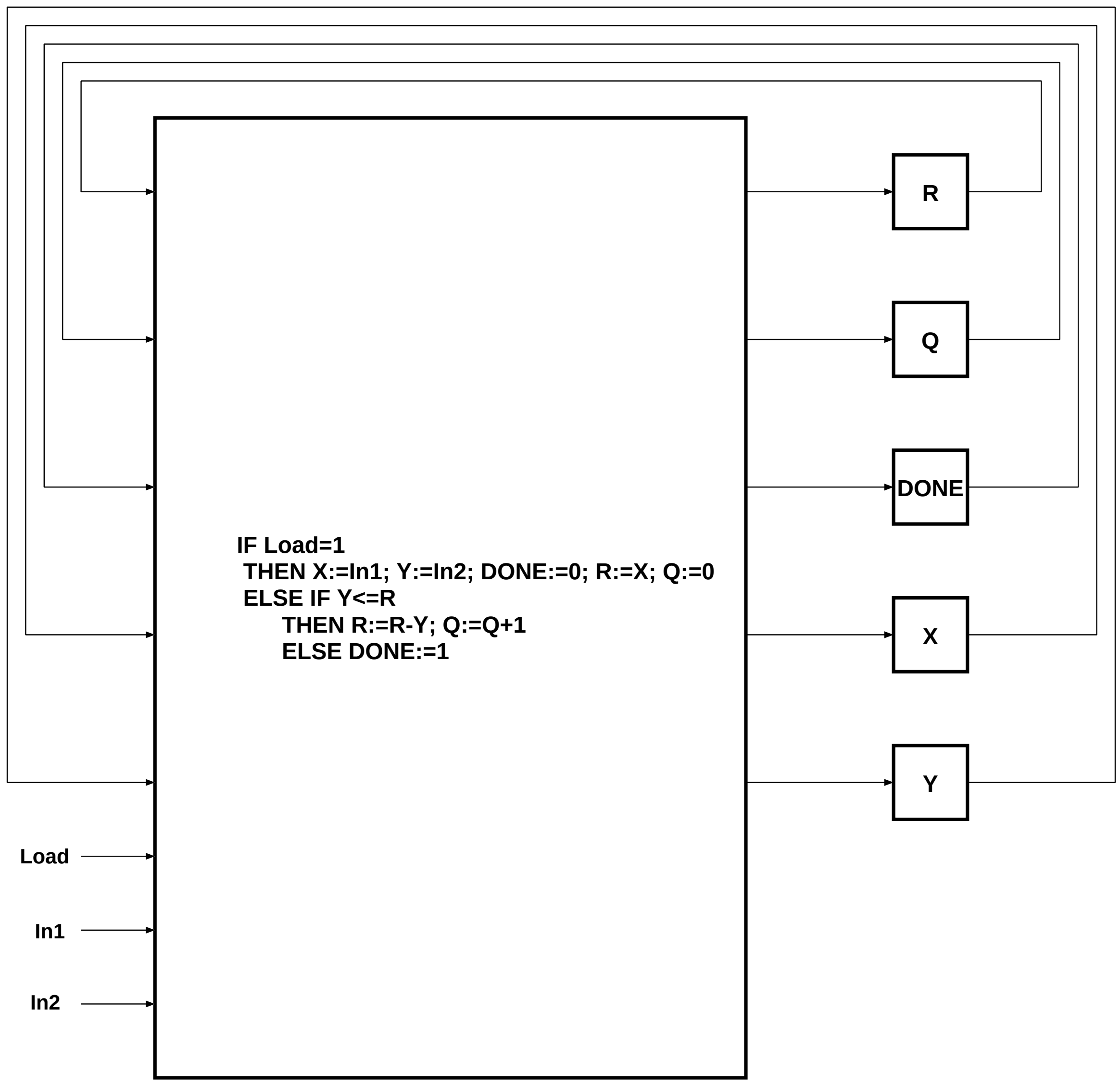
  - `In1`, `In2` **and** `Load` **are inputs whose value is determined by the environment (e.g. the user)**
  - `X`, `Y`, `Q`, `R` **and** `DONE` **are registers whose value is set by the program**

- **Environment sets the input `Load` to `1` to initialise registers**

- **To perform a division:**
  - `Load` **is set to** `0`
  - **and held at this value until** `DONE=1`
  - **so the environment must ensure that** `DONE=0` $\Rightarrow$ `Load=0`

FOREVER C

- **Each iteration step consists of**
  - **Circuit C computes new values of registers from current values and inputs**
  - **then updating the registers**



Programs as temporal statements

- **Would like a generalised Hoare Logic specification:**

$\vdash$ {*If environment ensures always that:* DONE=0 $\Rightarrow$ Load=0
and if Load *is set to* 1 *when:* In1 $= x \ \wedge$ In2 $= y$}

```
FOREVER
 IF Load=1
  THEN X:=In1; Y:=In2; DONE:=0; R:=X; Q:=0
  ELSE IF Y≤R THEN R:=R-Y; Q:=Q+1
                ELSE DONE:=1
```

{*Then x and y will be stored into* X *and* Y
and on the next cycle DONE *will be set to* 0
and sometime later DONE *will be be set to* 1
and X *and* Y *won't change until* DONE *is set to* 1
and when DONE *goes to* 1 *we have:* $x =$ R $+\ y \times$ Q}

- **Stuff in red needs Temporal Logic**

Brief history of temporal logic

- **1950s: philosophers invent temporal logic (A.N. Prior of Oxford)**
- **1970s: Burstall, Pnueli, Lamport use temporal logic for programs**
- **1980s: Emerson, Clarke and other introduce model checking**
- **1980s: hardware verification examples studied**
- **1990s: model checking catches on:
Intel hires many logicians for P7 verification. Uses STE.
Currently developing higher order logic tools ($reFL^{ect}$).**
- **1997: Amir Pnueli gets the Turing Award in recognition of his contribution to the applications of temporal logic**
- **2004: temporal notation for properties debated and standardised**
  - **semantics: CTL versus LTL**
  - **syntax: PSL and SVA 'aligned'**
- **2005 onwards: Assertion Based Verification (ABV) grows**
  - **dynamic checking of properties by simulation (e.g. used at ARM)**
  - **static checking by model checking**
- **2008: Clarke, Emerson & Sifakis get Turing prize for model checking**
- **2008: Clarke gets 2008 CADE Herbrand Award**
- **Note: work on formal methods leads to high prestige awards!**

Rest of the course

- **First look at 'raw' higher order logic for specification and verification**
  - temporal logic is a notation for specifying properties of traces
  - first look at reasoning directly about traces in higher order logic
- **Towards the end of the course we return to temporal logic**
  - look at its constructs
  - semantics via a shallow embedding in higher order logic
  - look at the 'Industry Standard' logic PSL
  - overview some key ideas for model checking temporal logic properties