# EENG 426/CPSC 459/ENAS 876
# Silicon Compilation

**Specifying Algorithms:**
**Communicating Hardware Processes**

Computer Systems Lab
http://csl.yale.edu/~rajit

Fall 2018

Yale AVLSI



## Communicating Hardware Processes

CHP = Communicating Hardware Processes

(based on Tony Hoare's CSP language)

CSP = Communicating Sequential Processes

- Assignment-based (imperative)
- No memory allocation
- No variable declaration (only shared/local)

Not a complete programming language.

Yale AVLSI



## Sequential part

**skip** statement that does nothing.

$x := E$ assignment statement

$x := \textbf{true} \equiv x\uparrow$
$x := \textbf{false} \equiv x\downarrow$
$x := a \vee b$
$x := c \wedge \neg d$
$y := y + 1$

Operationally:

- evaluate expression on the RHS
- assign the result of evaluation to the variable

Yale AVLSI



## Sequential part

Arrays are really an address-calculation mechanism.

$dmem[addr] := x$
$ins := imem[pc]$

Only use arrays when absolutely necessary!

Bit-fields are useful for extracting part of a variable

$$reg[ins_{31..27}] := reg[ins_{26..22}] + reg[ins_{21..17}]$$

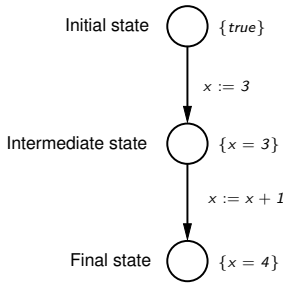
Yale AVLSI

## Composition

Sequential Composition:

$S_1 ; S_2$

Execute statement 1. Then execute statement 2.




Yale

AVLSI



## Control structures: selection

Selection Statement: Used to choose between different alternatives, like an IF statement.
Deterministic selection:

$$
\begin{array}{l}
[G_1 \longrightarrow S_1 \\
[]G_2 \longrightarrow S_2 \\
[]... \\
[]G_n \longrightarrow S_n \\
]
\end{array}
$$

Wait for one $G_i$ to be true. Execute corresponding $S_i$.
Restriction: Atmost one guard can be true.

Yale

AVLSI



## Control structures: selection

Non-deterministic selection Statement:

$$
\begin{array}{l}
[G_1 \longrightarrow S_1 \\
|G_2 \longrightarrow S_2 \\
|... \\
|G_n \longrightarrow S_n \\
]
\end{array}
$$

More than one $G_i$ can be true. Pick one.

Standard abbreviation:

$$[B] \equiv [B \longrightarrow \textbf{skip}]$$

Yale

AVLSI



## Examples

What do the following programs do?

$$
\begin{array}{l}
[\textbf{true} \longrightarrow x\uparrow \\
|\textbf{true} \longrightarrow x\downarrow \\
]
\end{array}
$$

$$[xi] ; xo\uparrow ; [\neg xi] ; xo\downarrow$$

$$
\begin{array}{l}
[x = 1 \longrightarrow y := 3 \\
[]x = 2 \longrightarrow y := 5 \\
[]x = 3 \longrightarrow y := 7 \\
]
\end{array}
$$

Yale

AVLSI

## Control structures: loops

Loops: Repeat actions while conditions are satisfied. A generalization of the WHILE loop.

Deterministic loop:

$$
\begin{array}{l}
*[G_1 \longrightarrow S_1 \\
\ [] G_2 \longrightarrow S_2 \\
\ [] \ldots \\
\ [] G_n \longrightarrow S_n \\
\ ]
\end{array}
$$

At most one $G_i$ can be true. Repeatedly execute until all guards are false.

Yale AVLSI



## Control structures: loops

Non-deterministic loop:

$$
\begin{array}{l}
*[G_1 \longrightarrow S_1 \\
\ | G_2 \longrightarrow S_2 \\
\ | \ldots \\
\ | G_n \longrightarrow S_n \\
\ ]
\end{array}
$$

More than one $G_i$ can be true.

Standard abbreviation:

$$*[S] \equiv *[\textbf{true} \longrightarrow S]$$

Yale AVLSI



## Examples

Add integers from one to ten:

$$
\begin{array}{l}
i := 1; \\
sum := 0; \\
*[i \leq 10 \longrightarrow sum := sum + i; i := i + 1]
\end{array}
$$

Euclid's algorithm:

$$
\begin{array}{l}
\{X > 0 \wedge Y > 0\} \\
x, y := X, Y; \\
*[x > y \longrightarrow x := x - y \\
\ [] x < y \longrightarrow y := y - x \\
\ ] \\
\{x = gcd(X, Y)\}
\end{array}
$$

Yale AVLSI

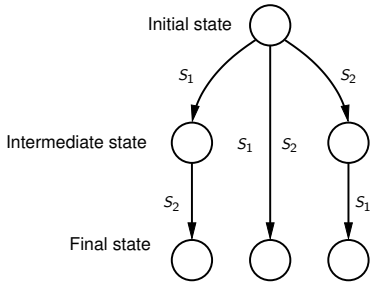


## Concurrency

Parallel composition:

$$S_1 \| S_2$$

Execute statements 1 and 2 in parallel.




Yale AVLSI

## Concurrency

What is the net effect of:

1. $x := 1 \parallel y := 2$
2. $x := 1 \parallel x := 2$
3. $x := x + 1 \parallel x := 3$

($x = 0$ initially)

Yale AVLSI



## Concurrency

1. $x = 1 \wedge y = 2$

Postulate: Concurrent activities of programs that do not share variables do not interfere with each another.

We're assuming a perfect physical world. Some effects that we're ignoring include:

- charge-sharing
- cross-talk
- ground bounce

Therefore, we have to ensure that these effects are in fact negligible!

Yale AVLSI



## Concurrency

2. *Software* : $x = 1 \vee x = 2$
   *Hardware* : $x = 1 \vee x = 2 \vee$ *bad stuff*

3. *Software* : $x = 1 \vee x = 3 \vee x = 4$
   *Hardware* : $x = 1 \vee x = 3 \vee x = 4 \vee$ *bad stuff*

No atomicity postulate for writing variables!!!

We use arbiters to solve the problem of interference.
When designing hardware, our task is to implement (in one form or another) the atomicity assumption that is commonly made by software!

Yale AVLSI



## Concurrency

What is the net effect of:

4. $x\uparrow \parallel y := x$
5. $x\uparrow \parallel [x]; y := x$
6. $*[\ x\uparrow; x\downarrow\ ] \parallel *[\ y\uparrow; y\downarrow\ ]$
7. $*[\ x\uparrow; x\downarrow\ ] \parallel *[\ [x]; y\uparrow; y\downarrow\ ]$

(initially $x$ and $y$ are false)

Yale AVLSI

## Concurrency

4. $x$

5. $x \wedge y$

Postulate: Reading a single boolean-valued variable is an atomic action.

Implementing such a read is tricky in the case when the variable being read is being modified at the same time. The implementation of such a read uses a synchronizer.

Yale AVLSI



## Concurrency

6. *infinite execution*

Postulate (weak fairness): We assume that actions from both processes get a chance to execute eventually. If an action from a process can execute, it will get to execute eventually.

7. *infinite execution*

In this case, it is possible that only a finite number of changes to $y$ occur.

An action must be enabled and stay enabled for it to get a chance to execute.

Yale AVLSI



## CPU example

$$
\begin{array}{l}
*[(i, pc) := (imem[pc], pc + 1); \\
\quad [offset(i.op) \longrightarrow (offset, pc) := imem[pc], pc + 1 \\
\quad []\neg offset(i.op) \longrightarrow \textbf{skip} \\
\quad ]; \\
\quad [alu(i.op) \longrightarrow (reg[i.z], f) := aluf(reg[i.x], reg[i.y], i.op, f) \\
\quad []ld(i.op) \longrightarrow reg[i.z] := dmem[reg[i.x] + reg[i.y]] \\
\quad []st(i.op) \longrightarrow dmem[reg[i.x] + reg[i.y]] := reg[i.z] \\
\quad []ldx(i.op) \longrightarrow reg[i.z] := dmem[offset + reg[i.y]] \\
\quad []stx(i.op) \longrightarrow dmem[offset + reg[i.y]] := reg[i.z] \\
\quad []lda(i.op) \longrightarrow reg[i.z] := offset + reg[i.y] \\
\quad []stpc(i.op) \longrightarrow reg[i.z] := pc + reg[i.x] \\
\quad []jmp(i.op) \longrightarrow pc := reg[i.y] \\
\quad []brch(i.op) \longrightarrow [cond(f, i.cc) \longrightarrow pc := pc + offset \\
\qquad\qquad\qquad\qquad []\neg cond(f, i.cc) \longrightarrow \textbf{skip} \\
\qquad\qquad\qquad\qquad ] \\
]]
\end{array}
$$

Yale AVLSI

*note the use of functions for decoding and alu operations.*