GEZEL: Basics
We will capture hardware designs into a language called GEZEL

Cycle-Based Bit-Parallel HW
Single-clock cycle digital logic design is called synchronous design
We first describe variables used in synchronous design

Consider a 3-bit counter
\[
\text{reg } a : \text{ns}(3); \quad // \text{a three-bit unsigned register}
\]
\[
\text{always} \{ \\
\quad a = a + 1; \quad // \text{each clock cycle, increment a} \\
\}
\]

This fragment specifies a three-bit register, called \(a\), which is incremented on each clock cycle

Given 3 bits, the register will count from 0 to 7, and wrap
Also, although the initial value of \(a\) is not specified, we will assume it is initialized to 0
GEZEL: Basics

The equivalent circuit for the statement $a = a + 1$

The `always` statement encapsulates the ONLY notion of time (updates occur on rising edge of clk), i.e., the stmt $a+1$ happens immediately after $a$ changes

Fig. 4.1 Equivalent Hardware for the three-bit counter
GEZEL: Basics

In GEZEL, we also have signals (like we do in VHDL), which means ’wire’

```plaintext
1 reg a : ns(3); // a three-bit unsigned register
2 sig b : ns(3); // a three-bit underscored signal
3 always {
4   b = a + 1;
5   a = b;
6 }
```

A signal instantly takes on the value of the expression assigned to it, i.e., \( b \) will instantly reflect the value of \( a+1 \)

Note the circuit diagram for this program is the same as the one shown earlier -- we’ve just given a name to the input of reg \( a \)

Like VHDL, a signal in GEZEL has no memory, with the same semantics, i.e., when used on the right-side, a signal will return the value assigned during that clk cycle

Thus, the lexical order of expressions has no meaning -- only the dataflow between reading/writing registers
GEZEL: Basics

Same behavior as previous

1 \texttt{reg a : ns}(3); // a three-bit unsigned register
2 \texttt{sig b : ns}(3); // a three-bit unsigned signal
3 \texttt{always} {
4 \hspace{1em} a = b;
5 \hspace{1em} b = a + 1;
6 \}

One important distinction between \textit{registers} and \textit{signals} (as you may recall)
  
  • When a register is used as an operand in an expression, it will return the value assigned to it in the \textbf{previous clk cycle}
  
  • When a signal is used as an operand in an expression, it will return the value assigned to it during the \textbf{current clk cycle}

Therefore, registers implement communication \textbf{across} clock cycles, while signals implement communication \textbf{within} a single clock cycle

Also remember that signals \textbf{cannot} have an initial value (no memory)

Therefore, a signal’s value is \textbf{undefined} when it is not assigned to
GEZEL: Basics

Illegal in GEZEL

• Use an undefined signal as an operand in an expression (generates simulator run-time error)

• Combinational feedback is also illegal

```plaintext
1 sig a : ns(3);
2 sig b : ns(3);
3 always {
4   a = b;
5   b = a + 1; // this is not a valid GEZEL program!
6 }
```

Precision and Sign in GEZEL

The wordlength and the sign of a register or signal are specified in their declaration

Some examples: a 4-bit unsigned register `a` and a 3-bit signed signal `b` (2’s complement format)

```plaintext
reg a : ns(4); // unsigned 4-bit value
sig b : tc(3); // signed 3-bit value
```
GEZEL: Basics

Registers and signals of different wordlengths can be combined in an expression

- The evaluation of an expression will never loose precision
- Assigning the result of an expression, or casting an expression type, will adjust the precision of the result

This code stores the value 12 in register $a$

```vhd
1 reg a : ns(4); // a four-bit unsigned number
2 sig b : ns(2); // a two-bit unsigned number
3 always {
4   b = 3;       // assign 0b(011) to b
5   a = b + 9;   // add 0b(11) and 0b(1010)
6 }
```

Evaluation rules are as follows

- The constant 3 is assigned to $b$ (011 is used for constant because it’s 2’s complement but signal is 2-bits, so only low order bits assigned)
- $a$ is assigned the sum of 9 (1001) and $b$ -- this is accomplished by sign-extending $b$ to 0011 (unsigned type case) to yield 12 (1100)
GEZEL: Basics

Beware:

1. `reg a : ns(6);` // a six-bit unsigned number
2. `sig b : tc(2);` // a two-bit signed number
3. `always {`
4. `b = 3;` // assign 0b(011) to b
5. `a = b - 3;` // subtract 0b(11) and 0b(011)
6. `}

After assigning the constant 3 to b, the value of b will be -1 (bit pattern 11)!
This generates -4 for b - 3 stmt, and assigns 111100 to a

But a is declared as unsigned, therefore this bit pattern is interpreted as 60!

Typecasting is allowed:

`(tc(1)) 1` // has the value -1
`(ns(3)) 15` // has the value 8
GEZEL: Hardware Mapping of Expressions

There is an equivalent hardware circuit for each expression involving signals and registers.

Arithmetic Operations
Addition (+), subtraction (-), multiplication (*) are commonly used in datapath hardware design, while division (/) and modulo (%) on the other hand are not.

Left-shift (<<) and right-shift (>>) are used to implement multiplication/division with powers of two.
Note that constant-shifts are translated into simple hardware wiring.

Bitwise Operations
Bitwise operators AND (&), OR (|), XOR (^) and NOT (~) have a direct equivalent to logic gates.

Operands of unequal length, they will be extended until they match.
GEZEL: Hardware Mapping of Expressions

Comparison Operations

All comparison operators return a single unsigned bit ($\text{ns}(1)$)

These operations use a subtractor to compare two numbers, and then use the sign & overflow flags of the result to evaluate the result of the comparison

Exact comparison (== or !=) can be done by matching the bitpattern of each operand

The comparison operations are implemented differently for signed and unsigned numbers (unlike arithmetic operations)

The bit pattern 111 is smaller than the pattern 001 for signed numbers, but is bigger for unsigned numbers

Bitvector Operations

Single bits, or a vector of several bits, can be extracted out of a word using the bit-selection operator

\begin{verbatim}
reg a : ns(5);
reg b : ns(1);
\end{verbatim}
GEZEL: Hardware Mapping of Expressions

```verilog
reg c : ns(2);
always {
    b = a[3];       // if a = 10111, then b = 0
    c = a[4:2];     // and a[4:2] = 101, so c = 01
}
```

Bit-concatenation (#)

```verilog
reg a : ns(5);
reg b : ns(1);
reg c : ns(2);
always {
    a = c # b;      // if b = 0, c = 11, then a = 00110
}
```

Selection

The ternary operator $a \ ? \ b : c$ is the equivalent notation for a multiplexer (same as C)
GEZEL: Hardware Mapping of Expressions

Indexed storage

Although there is no array construction operator in GEZEL, it is possible to capture lookup tables

```
lookup T : ns(12) = {0x223, 0x112, 0x990};
reg a : ns(12);
always {
    a = T(2); // a = 0x990
}
```

Organization and Precedence

Finally, brackets may be used to group expressions and change the evaluation order.

The default evaluation order is given by the following table, where higher number corresponds to a higher precedence.
GEZEL: Hardware Mapping of Expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
<th>Implementation</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>adder</td>
<td>4</td>
</tr>
<tr>
<td>Subtraction</td>
<td>−</td>
<td>subtractor</td>
<td>4</td>
</tr>
<tr>
<td>Unary Minus</td>
<td>−</td>
<td>subtractor</td>
<td>7</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>multiplier</td>
<td>5</td>
</tr>
<tr>
<td>Right-shift</td>
<td>&gt;&gt; (variable)</td>
<td>variable-shifter</td>
<td>0</td>
</tr>
<tr>
<td>Left-shift</td>
<td>&lt;&lt; (variable)</td>
<td>variable-shifter</td>
<td>0</td>
</tr>
<tr>
<td>Constant Right-shift</td>
<td>&gt;&gt; const</td>
<td>wiring</td>
<td>4</td>
</tr>
<tr>
<td>Constant Left-shift</td>
<td>&lt;&lt; const</td>
<td>wiring</td>
<td>4</td>
</tr>
<tr>
<td>Lookup Table</td>
<td>A (n)</td>
<td>random logic</td>
<td>10</td>
</tr>
<tr>
<td>And</td>
<td>&amp;</td>
<td>and-gate</td>
<td>2</td>
</tr>
<tr>
<td>Or</td>
<td></td>
<td></td>
<td>or-gate</td>
</tr>
<tr>
<td>Xor</td>
<td>^</td>
<td>xor-gate</td>
<td>3</td>
</tr>
<tr>
<td>Not</td>
<td>~</td>
<td>not-gate</td>
<td>8</td>
</tr>
<tr>
<td>Smaller-then</td>
<td>&lt;</td>
<td>subtractor</td>
<td>3</td>
</tr>
<tr>
<td>Bigger-then</td>
<td>&gt;</td>
<td>subtractor</td>
<td>3</td>
</tr>
<tr>
<td>Smaller-equal-then</td>
<td>&lt;=</td>
<td>subtractor</td>
<td>3</td>
</tr>
<tr>
<td>Bigger-equal-then</td>
<td>&gt;=</td>
<td>subtractor</td>
<td>3</td>
</tr>
<tr>
<td>Equal-to</td>
<td>==</td>
<td>comparator</td>
<td>3</td>
</tr>
<tr>
<td>Not-equal-to</td>
<td>!=</td>
<td>comparator</td>
<td>3</td>
</tr>
<tr>
<td>Bit Selection</td>
<td>[const]</td>
<td>wiring</td>
<td>9</td>
</tr>
<tr>
<td>Bit-vector Selection</td>
<td>[const:const]</td>
<td>wiring</td>
<td>9</td>
</tr>
<tr>
<td>Bit Concatenation</td>
<td>#</td>
<td>wiring</td>
<td>4</td>
</tr>
<tr>
<td>Type cast</td>
<td>(type)</td>
<td>wiring</td>
<td>6</td>
</tr>
<tr>
<td>Precedence ordering</td>
<td>( )</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>Selection</td>
<td>? :</td>
<td>multiplexer</td>
<td>1</td>
</tr>
</tbody>
</table>
GEZEL: Hardware Mapping of Expressions

Each expression involving registers, signals and operators from this table corresponds to a hardware datapath

Consider GCD, where two registers $m$ and $n$ are compared and the smallest one is subtracted from the largest during each clk cycle

```vhdl
1  reg m, n : ns(16);
2  always {
3     m = (m > n) ? (m - n) : m;
4     n = (n > n) ? (n - m) : m;
5  }
```

Consider a Linear Feedback Shift Register (LFSR), which is a shift register with a feedback loop created by XORing bits within the shift register

LFSRs are used for pseudo-random sequence generation

Can generate long non-repeating sequence of pseudorandom bits with a minimal amount of hardware
GEZEL: Hardware Mapping of Expressions

The feedback pattern is specified by a polynomial, e.g., one with \( p(x) = x^4 + x^3 + 1 \)

\[ \text{Fig. 4.2 Linear Feedback Shift Register for } p(x) = x^4 + x^3 + 1 \]

The resulting sequence of pseudorandom bits has a period of \( 2^n - 1 \) if the polynomial is selected in a special way (a maximum-length polynomial)

The shift register used to implement the LFSR must always contain at least one non-zero bit

Therefore, an LFSR must be initialized with a non-zero seed value, as shown by the multiplier in front of each register
GEZEL: Hardware Mapping of Expressions

This circuit is quite compact when written using word-level expressions

1. \texttt{reg shft : ns(4);}  
2. \texttt{sig shft\_new : ns(4);}  
3. \texttt{sig load : ns(1);}  
4. \texttt{sig seed : ns(4);}  
5. \texttt{always} \{ 
6. \hspace{1em} \texttt{shft\_new = (shft} \ll 1) \texttt{ | (shft[2] ^ shft[3]);}  
7. \hspace{1em} \texttt{shft} = \texttt{load} ? \texttt{seed} : \texttt{shft\_new;}  
8. \} 

Hardware Modules

A hardware module defines a level of hierarchy, and as in VHDL, hardware ports are used to communicate across levels of the hierarchy

![3-bit counter module](image)  

Fig. 4.3 Three-bit counter module
GEZEL: Hardware Modules

Only a single input, \textit{clr}, is defined, that enables a synchronous clear of the register

There is also a 3-bit output port, \textit{c}, that holds the current count value

GEZEL description

\begin{verbatim}
1 dp count(in clr : ns(1);  
2     out c   : ns(3)) {  
3     reg a : ns(3);  
4     always {  
5         a = clr ? 0 : a + 1;  
6         c = a;  
7     }  
8 }
\end{verbatim}

The \texttt{always} block is included in a \texttt{dp} (datapath), which defines a list of \texttt{in} and \texttt{out} ports

Ports are defined as signals and must be treated as such, i.e., exactly one assignment is allowed per clock, etc.
GEZEL: Hardware Modules

To define 3-bit counter as a component in a parent design, e.g., in a testbench

```
count
  c[0]   c[1]
  clr   c
```

**Fig. 4.4** Hardware equivalent of Listing 4.8

```
1 dp countrun {
2   sig clearit : ns(1);
3   sig cnt      : ns(3);
4   use count(clearit, cnt); // use keyword
5   always {
6       clearit = cnt[0] & cnt[1];
7       $display("cnt = ", cnt);
8   }
9 }
```
GEZEL: Hardware Modules

The countrun module has no inputs nor outputs (it’s a testbench)

Note the use of the $display stmt, which is a simulator directive to print the value of a signal or register.

Once a module has been included inside another, it cannot be included again -- however, it is easy to create a duplicate by means of a cloning statement.

To create three three-bit counters, count0, count1 and count2:

```vhls
1 dp count0(in clr : ns(1);
2          out c : ns(3)) {
3    reg a : ns(3);
4    always {
5        a = clr ? 0 : a + 1;
6        c = a;
7    }
8 }
9 dp count1 : count0
10 dp count2 : count0
```