# Appendix A: Verilog Cheat Sheet

## Numbers and constants

Example: 4-bit constant 10 in binary, hex and in decimal: 4’b1010 == 4’ha -- 4’d10

(numbers are unsigned by default)

Concatenation of bits using {}

4’b1011 == {2’b10 , 2’b11}

Constants are declared using parameter:

parameter myparam = 51

## Operators

Arithmetic: and (+), subtract (-), multiply (*), divide (/) and modulus (%) all provided.

Shift: left (<<), shift right (>>)

Relational ops: equal (==), not-equal (!=), less-than (<), less-than or equal (<=), greater-than (>), greater-than or equal (>=).

Bitwise ops: and ( & ), or ( | ), xor ( ^ ), not ( ~ )

Logical operators: and (&&) or (||) not (!) note that these work as in C, e.g. (2 && 1) == 1

Bit reduction operators: [n] n=bit to extract

Conditional operator: ? to multiplex result

Example: (a==1)? funcif1 : funcif0

The above is equivalent to:

```
  ((a==1) && funcif1)
  || ((a!=1) && funcif0)
```

## Registers and wires

Declaring a 4 bit wire with index starting at 0:

wire [3:0] w;

Declaring an 8 bit register:

reg [7:0] r;

Declaring a 32 element memory 8 bits wide:

reg [7:0] mem [0:31]

Bit extract example:

r[5:2] returns 4 bits between pos 2 to 5 inclusive

## Assignment

Assignment to wires uses the assign primitive outside an always block, e.g.:

assign mywire = a & b

Registers are assigned to inside an always block which specifies where the clock comes from, e.g.:

```
always@(posedge myclock)
   cnt = cnt + 1;
```

## Blocking vs. unblocking assignment <= vs. =

The <= assignment operator is non-blocking (i.e. if use in an always@(posedge) it will be performed on every positive edge. If you have many non-blocking assignments they will all be updated in parallel. The <= operator must be used inside an always block – you can’t use it in an assign statement.

The blocking assignment operator = can be used in either an assign block or an always block. But it causes assignments to be performed in sequential order. This tends to result in slower circuits, so avoid using it (especially for synthesized circuits) unless you have to.

## Case and if statements

Case and if statements are used inside an always block to conditionally update state. e.g.:

```
always @(posedge clock)
  if (add1 && add2) r <= r+3;
  else if (add2) r <= r+2;
  else if(add1) r <= r+1;
```

Note that we don’t need to specify what happens when add1 and add2 are both false since the default behavior is that r will not be updated. Equivalent function using a case statement:

```
always @(posedge clock)
  case({add2,add1})
  2’b11  : r <= r+3;
  2’b10  : r <= r+2;
  2’b01  : r <= r+1;
  default: r <= r;
endcase
```

## Module declarations

Modules pass inputs, outputs as wires by default.

```
module ModName (
  output reg [3:0] result,  // register output
  input [1:0] bitsin,  input clk, inout bidirectnl  );
 … code …
endmodule
```

## Verilog Simulation / ISIM commands

```
$display ("a string to display");
$monitor ("like printf. Vals: %d %b", decv,bitv);
#100   // wait 100ns or simulation moments
$finish  // end simulation
```

# Appendix B: ARM Assembly Language Cheat Sheet

**Memory access instructions**
```
LDR Rd, [Rn]            ; load 32-bit number at [Rn] to Rd
LDR Rd, [Rn,#off]       ; load 32-bit number at [Rn+off] to Rd
STR Rt, [Rn]            ; store 32-bit Rt to [Rn]
STR Rt, [Rn,#off]       ; store 32-bit Rt to [Rn+off]
PUSH {Rt}               ; push 32-bit Rt onto stack
POP  {Rd}               ; pop 32-bit number from stack into Rd

MOV{S} Rd, <op2>        ; set Rd equal to op2
MOV Rd, #im16           ; set Rd equal to im16, im16 is 0 to 65535
MVN{S} Rd, <op2>        ; set Rd equal to -op2
```

**Branch instructions**
```
B label                 ; branch to label Always
BEQ label               ; branch if Z == 1 Equal
BNE label               ; branch if Z == 0 Not equal
BCS label               ; branch if C == 1 Higher or same, unsigned ≥
BHS label               ; branch if C == 1 Higher or same, unsigned ≥
BCC label               ; branch if C == 0 Lower, unsigned <
BLO label               ; branch if C == 0 Lower, unsigned <
BMI label               ; branch if N == 1 Negative
BPL label               ; branch if N == 0 Positive or zero
BVS label               ; branch if V == 1 Overflow
BVC label               ; branch if V == 0 No overflow
BHI label               ; branch if C==1 and Z==0 Higher, unsigned >
BLS label               ; branch if C==0 or Z==1 Lower or same, unsigned ≤
BGE label               ; branch if N == V Greater than or equal, signed ≥
BLT label               ; branch if N != V Less than, signed <
BGT label               ; branch if Z==0 and N==V Greater than, signed >
BLE label               ; branch if Z==1 or N!=V Less than or equal, signed ≤
BX Rm                   ; branch indirect to location specified by Rm
BL label                ; branch to subroutine at label
BLX Rm                  ; branch to subroutine indirect specified by Rm
```

**Logical instructions**
```
AND{S} {Rd,} Rn, <op2>  ; Rd=Rn&op2 (op2 is 32 bits)
ORR{S} {Rd,} Rn, <op2>  ; Rd=Rn|op2 (op2 is 32 bits)
EOR{S} {Rd,} Rn, <op2>  ; Rd=Rn^op2 (op2 is 32 bits)
BIC{S} {Rd,} Rn, <op2>  ; Rd=Rn&(~op2) (op2 is 32 bits)
ORN{S} {Rd,} Rn, <op2>  ; Rd=Rn|(~op2) (op2 is 32 bits)
LSR{S} Rd, Rm, Rs       ; logical shift right Rd=Rm>>Rs (unsigned)
LSR{S} Rd, Rm, #n       ; logical shift right Rd=Rm>>n (unsigned)
ASR{S} Rd, Rm, Rs       ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S} Rd, Rm, #n       ; arithmetic shift right Rd=Rm>>n (signed)
LSL{S} Rd, Rm, Rs       ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n       ; shift left Rd=Rm<<n (signed, unsigned)
```

**Arithmetic instructions**
```
ADD{S} {Rd,} Rn, <op2>   ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12   ; Rd = Rn + im12, im12 is 0 to 4095
SUB{S} {Rd,} Rn, <op2>   ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12   ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2>   ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12   ; Rd = im12 – Rn
CMP Rn, <op2>            ; Rn – op2 sets the NZVC bits
MUL{S} {Rd,} Rn, Rm      ; Rd = Rn * Rm signed or unsigned
```

**Notes**

```
Ra Rd Rm Rn Rt represent 32-bit registers
value any 32-bit value: signed, unsigned, or address
{S} if S is present, instruction will set condition codes
#im12 any value from 0 to 4095
#im16 any value from 0 to 65535
{Rd,} if Rd is present Rd is destination, otherwise Rn
#n any value from 0 to 31
#off any value from -255 to 4095
label any address within the ROM of the microcontroller
op2 the value generated by <op2>
```