Introduction to High-level Synthesis

LISTA/UFSC

Prof. Dr. Antônio Augusto Fröhlich
Tiago Rogério Mück

http://www.lisha.ufsc.br/~guto
What is HLS?

Example:
- High-level algorithm to multiply an array by an factor and accumulate the result

```c
int mult_acc_array(int input[SIZE], int factor) {
    int result = 0;
    for (int i = 0; i < SIZE; ++i) {
        result += input[i] * factor;
    }
    return result;
}
```
What is HLS?

- Possible RTL implementation:

```verilog
wire [31:0] in[0:SIZE];
wire [31:0] factor;
reg [63:0] out;

integer i;
reg [63:0] aux;
always@(posedge clk)
begin
    aux = 0;
    for(i = 0; i < SIZE; i = i + 1)
        aux = aux + (factor * in[i]);
    out <= aux;
end
```
What is HLS?

- Possible RTL implementation:

```verilog
wire [31:0] in[0:SIZE];
wire [31:0] factor;
reg [63:0] out;
integer i;
reg [63:0] aux;
always@(posedge clk)
begin
    aux = 0;
    for(i = 0; i < SIZE; i = i + 1)
        aux = aux + (factor * in[i]);
    out <= aux;
end
```

- So, HLS is not:
  - Use of high-level constructs, operators and data types
    - +
    - *
    - for
    - while
    - integer
    - …
  - Those are considered valid RTL constructs and can be easily synthesized
However, the same algorithm can have many different RTL implementations.

```verilog
type integer i;
reg [63:0] aux;
always @(posedge clk)
begin
    aux = 0;
    for(i = 0; i < SIZE; i = i + 1)
        aux = aux + (factor * in[i]);
end

reg [31:0] index = 0;
always @(posedge clk)
begin
    out <= out + (factor * in[index]);
    index <= index + 1;
end

reg [63:0] mult_result;
reg [31:0] index = 0;
always @(posedge clk)
begin
    mult_result <= factor * in[index];
    out <= mult_result + out;
    index <= index + 1;
end
```
Full parallel implementation

- All iterations are executed in parallel in the same clock cycle

```
integer i;
reg [63:0] aux;
always@(posedge clk)
begin
    aux = 0;
    for(i = 0; i < SIZE; i = i + 1)
        aux = aux + (factor * in[i]);
    out <= aux;
end
```
Serial implementation

- Iterations executed sequentially in different clock cycles
- Takes 4 cycles to compute a new result
- Critical path is shorter → Higher clock frequency
- Less area

```verilog
reg [31:0] index = 0;
always@(posedge clk) begin
    out <= out + (factor * in[index]);
    index <= index + 1;
end
```
Pipelined implementation

- Multiplication and addition implemented in different cycles
- Critical path is even shorter → Higher clock frequency

```verilog
reg [63:0] mult_result;
reg [31:0] index = 0;
always@(posedge clk) begin
    mult_result <= factor * in[index];
    out <= mult_result + out;
    index <= index + 1;
end
```
What is HLS?

- The same algorithm can be implemented using many different microarchitectures.

- So, high-level synthesis is the (semi) automatic generation of a HW microarchitecture from a high-level algorithmic description.
High-level synthesis

```c
int mult_acc_array(int input[SIZE], int factor) {
    int result = 0;
    for (int i = 0; i < SIZE; ++i)
        result += input[i] * factor;
    return result;
}
```
HLS process

- **Algorithmic level**
  - Scheduling
  - Loop optimizations
    - Loop unrolling
    - Pipelining
- **Structural level**
  - Interface synthesis
  - Hierarchy, classes, functions, etc
Scheduling

- Scheduling is the process of translating an untimed algorithm into a timed algorithm
  - The clock is added to the system and operations are scheduled among clock cycles
- The scheduling is performed according to the system constraints
  - Target clock frequency
  - Target technology (FPGA, ASIC, etc)
  - Synthesis goal
    - Speed
    - Area
    - Power consumption
Scheduling (cont.)

Example

```c
int mult_acc_array(int input[SIZE], int factor) {
    int result = 0;
    for (int i = 0; i < SIZE; ++i)
        result += input[i] * factor;
    return result;
}
```

* and + on 1 cycle

```c
... for (int i = 0; i < SIZE; ++i) {
    wait(posedge clk)
    result += input[i] * factor;
}
...```

* and + on 2 cycles

```c
... for (int i = 0; i < SIZE; ++i) {
    wait(posedge clk)
    mult = input[i] * factor;
    wait(posedge clk)
    result += mult;
}
...```
Loop unrolling

- Loops are unrolled in order to execute its iterations in parallel
  - Increases HW throughput at the cost of extra resources

- Partial unrolling
  - Loop bounds must be always known
  ```
  ... wait(posedge clk)
  for (int i = 0; i < 2; ++i)
    result += input[i] * factor;
  ```

- Full unrolling
  - Possible only if there is no data dependency between iterations
  ```
  ... wait(posedge clk)
  for (int i = 2; i < 4; ++i)
    result += input[i] * factor;
  ```
Pipelining

- Sometimes a serial implementation is too slow
- And it is not possible to fully unroll loops due to data dependencies and/or area constraints
- Operations can be “pipelined”

Example: RISC pipeline
Pipelining (cont.)

- Example

- Pipelining using 1 mult. and 1 adder

C 1 | C 2 | C 3 | C 4 | C 5
MULT | ACC | MULT | ACC | MULT

... wait(posedge clk)
mult = input[ 0 ] * factor;
result = 0;
wait(posedge clk)
result += mult;
mult = input[ 1 ] * factor;
wait(posedge clk)
result += mult;
mult = input[ 2 ] * factor;
wait(posedge clk)
result += mult;
mult = input[ 3 ] * factor;
wait(posedge clk)
result += mult;
...

...
More HLS - Interface synthesis

- Hardware interface is (semi) automatically generated from function calls or high-level channels (e.g. SystemC `sc_fifo`, `sc_bus`, etc)

```plaintext
void mult_acc_array(
    int input[SIZE],
    int factor,
    int &result);
```

```plaintext
module mult_acc_array(
    input clk,
    input rst,
    input [(32*SIZE)-1:0] in,
    input [31:0] factor,
    output [63:0] result,
    input d_available,
    output d_ready
    );
```

- **Clock/Reset**
  - Clock/Reset

- **Handshaking signals**
  - Handshaking signals
  - High when new data is available
  - High when ready to accept new data
More HLS – Memory structures

- (semi) Automatic mapping of memory structures

int vals[1024]

Register bank

RAM memory

Single port

Dual port

- The actual mapping will depend on (or influence) the schedule
Study case - FIR implementation

- **FIR (Finite Impulse Response) filter**
  - A type of a digital signal processing filter whose output is a weighted sum of the current and a finite number of previous values of the input

\[ y[n] = b_0 x[n] + b_1 x[n-1] + \ldots + b_N x[n-N] \]

- **Applications**
  - Low/high/band-pass filters
  - Signal aliasing
  - etc
Study case - FIR implementation

- FIR filter synthesis using CatapultC