POLITECNICO DI MILANO

Compiler Based Optimizations, Tuning and Customization of Generated Accelerators

Tutorial @ FPL Conference 2017 – Ghent – Belgium

Marco Lattuada

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
marco.lattuada@polimi.it

Outline

- Integration of generated accelerators
- Tuning accelerators by means of optimizations
- Math support

Reset types and values

- Internal status of accelerators can be reset
  - Accelerators exposes a reset signal
- Register reset type:
  - no (default)
  - async
  - sync
- Reset level:
  - low (default)
  - high
- Example:

```
--reset-type=sync -reset-level=high
```

Registered inputs

- A dedicated port is created for scalar parameters of each module function
- Generated modules expect stable inputs
  - If inputs are not stable, they can be registered
- Registered inputs:
  - auto (default) – inputs are registered olnly for shared functions
  - yes
  - no

```
--registered-inputs=<value>
```

FSM-encoding

- Different types of encoding can be used in Finite State Machine
  - one-hot
  - binary
- Default: best encoding for logic synthesis tool
  - Vivado: one-hot
  - Other tools: binary

```
--fsm-encoding=<value>
```

Improve Area/Performance of generated accelerators

- Performance and/or area of the generated accelerators can be improved by tuning the design flow
  - GCC optimizations
  - Bambu IR optimizations
  - Bambu HLS algorithms
- Best design flow for every accelerator does not exist
  - Trade off between area and performance
  - Effects of the single optimizations can be different on the single accelerators
- Default:
  - Balanced area/performance trade off

Used example

- aes from CHStone suite
  - Yuko Hara, Hiroyuki Tomiyama, Shinya Honda and Hiroaki Takada, "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis", *Journal of Information Processing*, Vol. 17, pp.242-254, (2009).
- Target device:
  - xc7z020-1clg484-VVD (Zynq with Vivado)
- Target clock period:
  - 10 ns

Translation flow

- C→HDL without optimizations
  - GCC optimizations are (mostly) disabled
  - Bambu IR optimizations are (mostly) disabled

```
-O0 --cfg-max-transformations=0 --no-chaining
```

- Can be exploited only when bambu is compiled with development support
- Useful for debugging

GCC Optimizations

- Only GCC target independent optimizations are considered
- -O3 is not necessarily the best choice
  - Can improve performances
  - Can increment area
- User can tune this part of the flow:
  - Selecting optimization level:

```
-O0 or -O1 or -O2 or -O3 or -Os
```

  - Enabling/disabling single GCC optimization:

```
-f<optimization> -fno-<optimization>
```

  - Tuning gcc parameters: --para

```
--param <name>=<value>
```

Effect of GCC Optimizations

- Results refer to other Bambu options set to default value

Opts	Cycles	Luts
O0	15764	11675
O1	7892	11052
O2	4679	10276
O3	3854	15679
O3 vectorize	3816	38553
O3 all inline	1327	13550

Bambu IR Analysis

- Collect information used by IR optimizations and High Level Synthesis
- Data flow analysis
  - Scalar: based on SSA
  - Aggregates: exploit GCC+Bambu alias analysis
- Graphs Computation
  - Call Graph, CFG, DFG, …
- Loops identification
- Bit Value Analysis
  - Compute for each SSA which bit are used and which bit are fixed

Bambu IR Optimizations

- Applied before HLS to the IR produced by GCC
- Two type of optimizations
  - Single instruction optimizations
  - Multiple instruction optimizations
  - Restructuring of Control Flow Graph
  - Fixing IR
- Sequences of optimizations can be applied multiple times
  - Fixed point iteration optimization flow

Single Instruction Optimization

- IR lowering – make single instructions more suitable to be implemented on FPGA
  - Expansion of multiplication by constant
  - Expansion of division by constant
  - Etc.
- Bit Value Optimization
  - Shrink operations to the only significant bits

Multiple Instruction Optimization

- Common Subexpression Elimination
- Dead Code Elimination
- Extract pattern (e.g., three input sum)
- LUT transformations
  - Merging multiple Boolean operations into a single LUT-based operation
- Cond Expr Restructuring

Restructuring of Control Flow Graph

- Speculation
- Code motion
- Merging of conditional branch
  - Creation of multiple target branch
- Basic Block Removal
  - Empty
  - Last

System of Difference Constraints

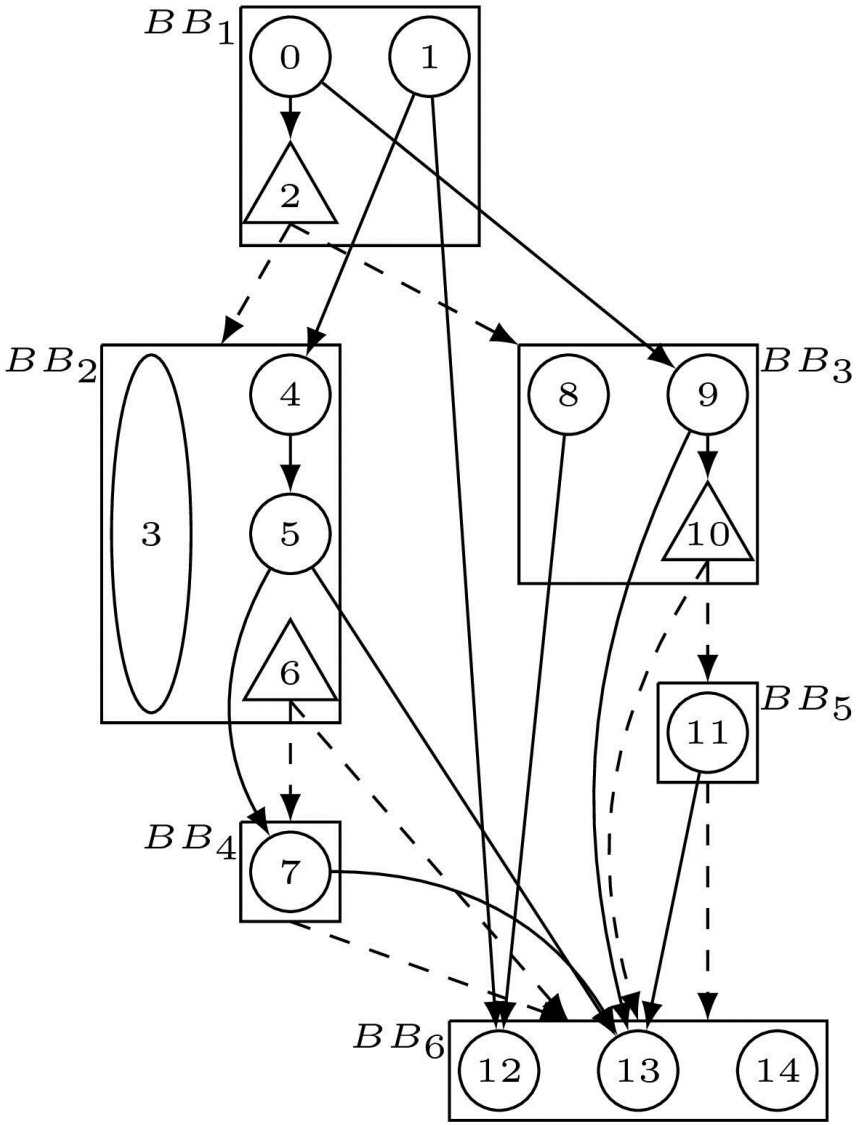
- Global scheduling based on ILP formulation
- Results are exploited to perform
  - Speculation
  - Code Motion

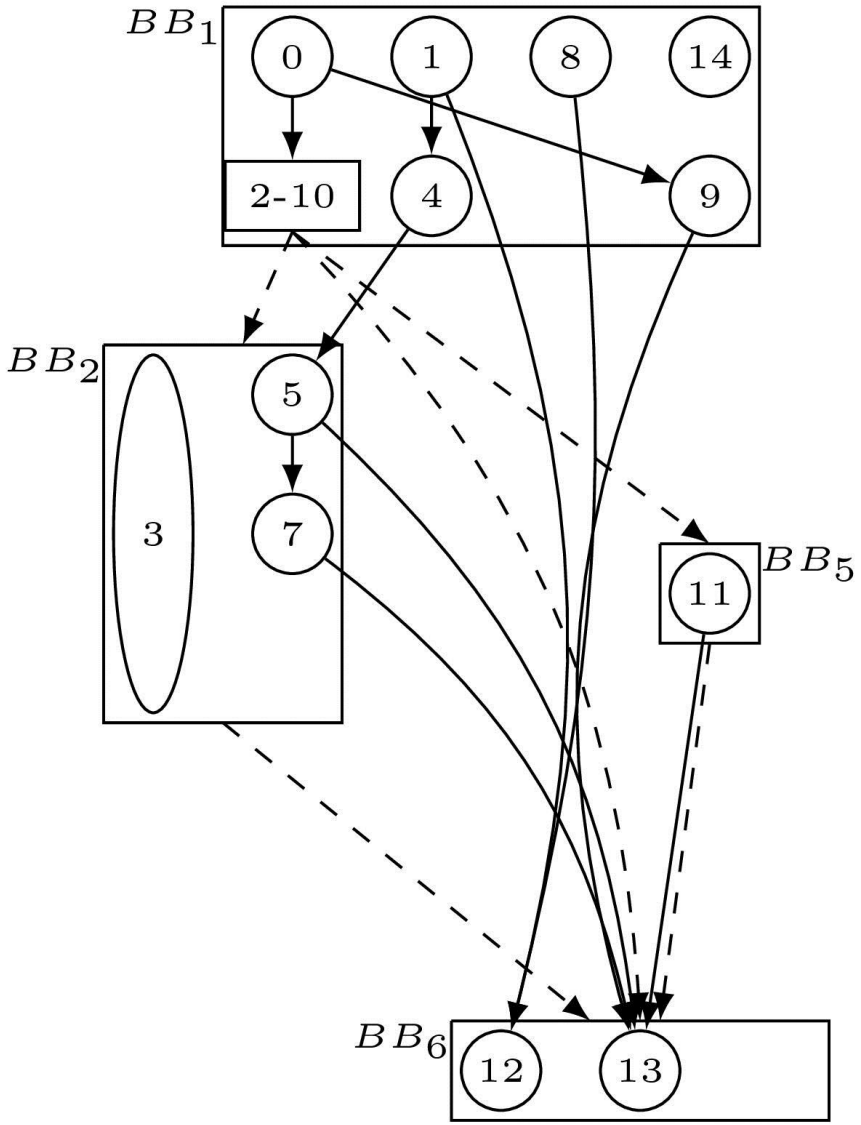
+ Improve performances of accelerators

- Potentially increment area of accelerators

- Increase High Level Synthesis time

IR optimizations: Example





Fixing IR

- Struct assignment
  - Replaced with memcpy call
- Floating point operations
  - Replaced with function calls
- Integer divisions
  - Replaced with function calls

Experimental setup

- Predefined design flows

```
--experimental-setup=<setup>
```

BAMBU-AREA: optimized for area

BAMBU-PERFORMANCE: optimized for performances

BAMBU-BALANCED: optimized for trade-off area/performance

BAMBU-AREA-MP, BAMBU-PERFORMANCE-MP, BAMBU-BALANCED-MP: enable support to true dual port memories

Default: BAMBU-BALANCED-MP

Constraints

- Bambu assumes infinite resources during High Level Synthesis
  - Produced solutions may not fit in the target device
- Area of generated solutions can be indirectly controlled by means of **constraints**
- User can constraint the number of available functional units in each function
  - E.g.: fix the number of available multiplier in each function
- Constraints are set by means of *XML file*

Example of constraints file


```
<?xml version="1.0"?>
  <constraints>
    <HLS_constraints>
      <tech_constraints fu_name="mult_expr_FU"
                        fu_library="STD_FU" n="8"/>
    </HLS_constraints>
  </constraints>
```

Integer Division Algorithms

- You can control how to implement integer divisions:

```
--hls-div=<implementation>
```

- Available implementations:
  - `none`: HDL based pipeline restoring division
  - **`nr1`** (default): C-based non restoring division with unrolling factor equal to 1
  - `nr2`: C-based non restoring division with unrolling factor equal to 2
  - `NR`: C-based Newton-Raphson division
  - `as`: C-based align divisor shift dividentd method

Floating point support

- Possible ways of implementing floating point ops:
  - Softfloat (default): customized faithfully rounded (nearest even) version of soft based implementation

```
--soft-float
```

  - Softfloat-subnormal: soft based implementation with support to subnormal

```
--softfloat-subnormal
```

  - Softfloat GCC: GCC soft based implementation

```
--soft-fp
```

  - Flopoco generated modules

```
--flopoco
```

Libm versions

- Bambu exploits High Level Synthesis to generate accelerators implementing libm functions
- Two different versions of libm are available
  1. Faithfully rounding (default)
  2. Classical libm built integrating existing libm source code from glibc, newlib, uclibc and musl libraries.
     - Worse performances and area