Efficient Looping Units for FPGAs

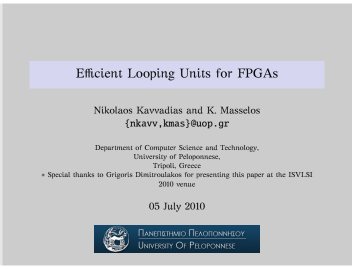
Nikolaos Kavvadias and K. Masselos
{nkavv,kmas}@uop.gr

Department of Computer Science and Technology,
University of Peloponnese,
Tripoli, Greece
∗ Special thanks to Grigoris Dimitroulakos for presenting this paper at the ISVLSI 2010 venue

05 July 2010

2010-06-28

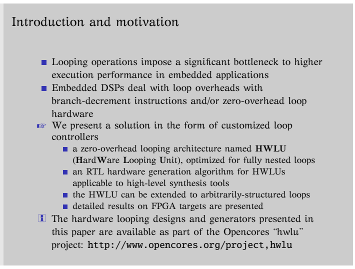
Efficient Looping Units for FPGAs



- No additional comments

Introduction and motivation

- Looping operations impose a significant bottleneck to higher execution performance in embedded applications
- Embedded DSPs deal with loop overheads with branch-decrement instructions and/or zero-overhead loop hardware
- ☞ We present a solution in the form of customized loop controllers
  - a zero-overhead looping architecture named **HWLU** (**Hard****W**are **L**ooping **U**nit), optimized for fully nested loops
  - an RTL hardware generation algorithm for HWLUs applicable to high-level synthesis tools
  - the HWLU can be extended to arbitrarily-structured loops
  - detailed results on FPGA targets are presented
- ℹ The hardware looping designs and generators presented in this paper are available as part of the Opencores “hwlu” project: `http://www.opencores.org/project,hwlu`

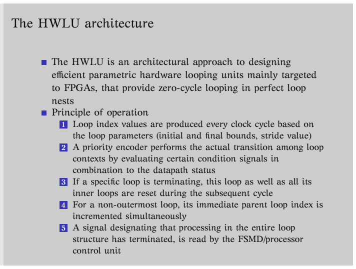

Introduction and motivation

- Looping operations impose a significant bottleneck to higher execution performance in embedded applications
- Embedded DSPs deal with loop overheads with branch-decrement instructions and/or zero-overhead loop hardware
- ☞ We present a solution in the form of customized loop controllers
  - a zero-overhead looping architecture named **HWLU** (**Hard****W**are **L**ooping **U**nit), optimized for fully nested loops
  - an RTL hardware generation algorithm for HWLUs applicable to high-level synthesis tools
  - the HWLU can be extended to arbitrarily-structured loops
  - detailed results on FPGA targets are presented
- ℹ The hardware looping designs and generators presented in this paper are available as part of the Opencores “hwlu” project: `http://www.opencores.org/project,hwlu`

- Contemporary general-purpose processor (ARM, MIPS32) and DSP architectures present architectural characteristics suitable to portable platforms. More and more often, embedded RISC/DSPs involve customized features to data-dominated domains, where the most performance-critical computations occur in various forms of nested loops.
- Following this trend, they provide better means for the execution of loops, by surpassing the significant overhead of the loop overhead instructions (the required instructions to initiate a new iteration of the loop)
- Soft-cores (MicroBlaze, Nios-II, LEON3) are a particular processor class aiming FPGAs
- These processors lack any looping hardware that would speed up looping operations
- We present the HWLU architecture, supported by an open-source generation tool

The HWLU architecture

- The HWLU is an architectural approach to designing efficient parametric hardware looping units mainly targeted to FPGAs, that provide zero-cycle looping in perfect loop nests
- Principle of operation
  1. Loop index values are produced every clock cycle based on the loop parameters (initial and final bounds, stride value)
  2. A priority encoder performs the actual transition among loop contexts by evaluating certain condition signals in combination to the datapath status
  3. If a specific loop is terminating, this loop as well as all its inner loops are reset during the subsequent cycle
  4. For a non-outermost loop, its immediate parent loop index is incremented simultaneously
  5. A signal designating that processing in the entire loop structure has terminated, is read by the FSMD/processor control unit

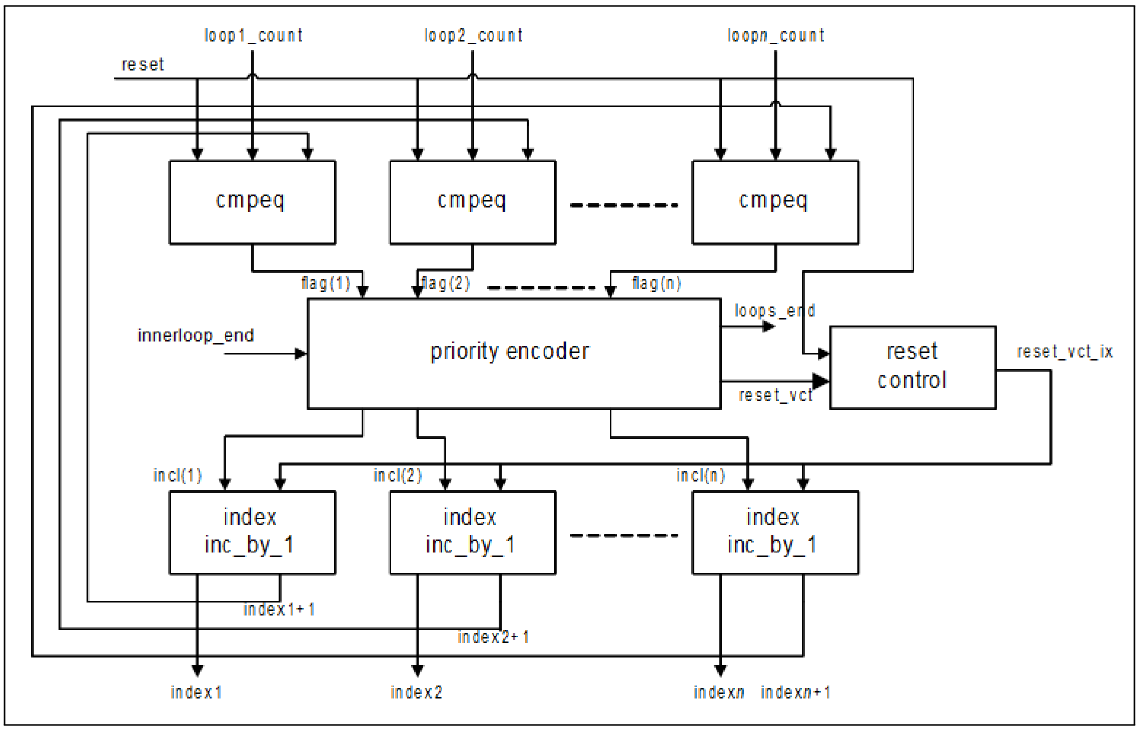


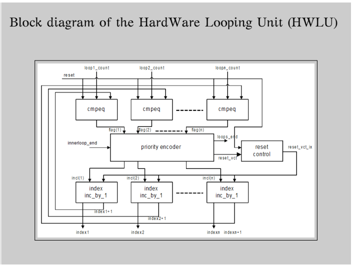
The HWLU architecture

- The HWLU is an architectural approach to designing efficient parametric hardware looping units mainly targeted to FPGAs, that provide zero-cycle looping in perfect loop nests
- Principle of operation
  1. Loop index values are produced every clock cycle based on the loop parameters (initial and final bounds, stride value)
  2. A priority encoder performs the actual transition among loop contexts by evaluating certain condition signals in combination to the datapath status
  3. If a specific loop is terminating, this loop as well as all its inner loops are reset during the subsequent cycle
  4. For a non-outermost loop, its immediate parent loop index is incremented simultaneously
  5. A signal designating that processing in the entire loop structure has terminated, is read by the FSMD/processor control unit

- A major advantage of the HWLU is that successive last iterations of nested loops are performed in a single cycle
- The HWLU can be useful in the case that all data processing in context of the nested loop structure is performed in the inner loop. This is rather often in multidimensional signal processing kernels such as performance-critical code in image coding and video compression standards

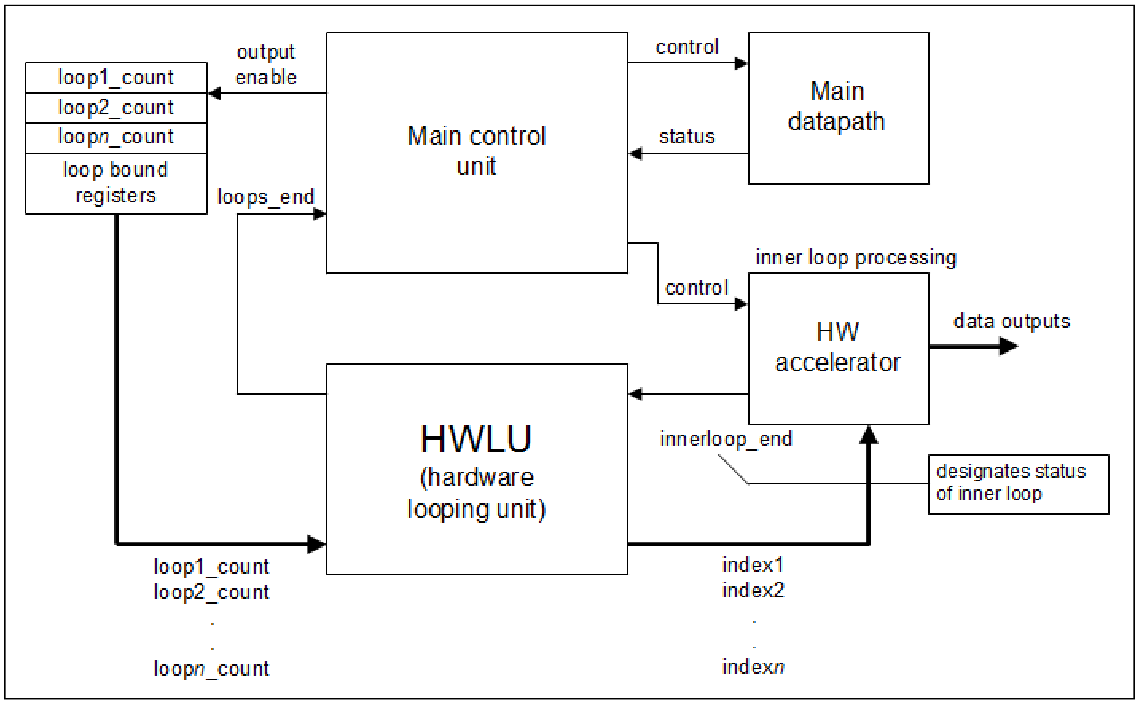
Block diagram of the HardWare Looping Unit (HWLU)

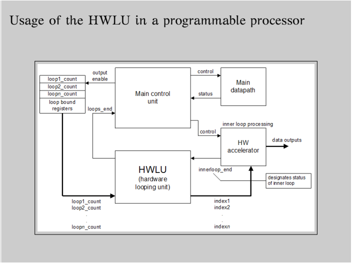




- Loop index values are produced every clock cycle based on the loop bound values for each nesting level
- In the following cycle of a last iteration for a specific loop, the loop index is reset to its initial value
- The priority encoder accepts the equality comparators (`cmpeq`) outputs (bitwise flag signals) and an external signal from the datapath (`innerloop_end`). This signal is produced by the corresponding hardware module that performs the inner loop operations, which may be a custom unit
- If a specific loop is terminating, this loop as well as all its inner loops are reset during the subsequent cycle by the priority encoder. For a non-outermost loop, its immediate parent loop index is incremented. If none of the loops is terminating, then the inner loop is incremented. Signal `innerloop_end` guards this increment operation
- Finally, signal `loops_end` designates that processing in the entire loop structure has terminated

Usage of the HWLU in a programmable processor

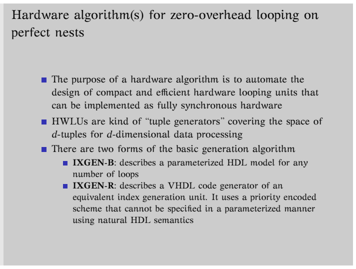




- This figure indicates a possible design of an HWLU-aware control unit used in a programmable processor
- Assume that the register architecture of the processor is partitioned, so that the loop index registers are stored into dedicated registers
- Control-dominated segments of the user program are implemented in the main datapath
- When appropriate, the main control unit activates the hardware acceleration datapath unit that performs all inner-loop processing
- When its operation terminates, the HWLU is acknowledged through the `innerloop_end` asynchronous flag
- On an active `loops_end` signal, which occurs when the loop structure is exited, the main control unit pauses the HWLU

Hardware algorithm(s) for zero-overhead looping on perfect nests

- The purpose of a hardware algorithm is to automate the design of compact and efficient hardware looping units that can be implemented as fully synchronous hardware
- HWLUs are kind of “tuple generators” covering the space of d-tuples for d-dimensional data processing
- There are two forms of the basic generation algorithm
  - **IXGEN-B**: describes a parameterized HDL model for any number of loops
  - **IXGEN-R**: describes a VHDL code generator of an equivalent index generation unit. It uses a priority encoded scheme that cannot be specified in a parameterized manner using natural HDL semantics

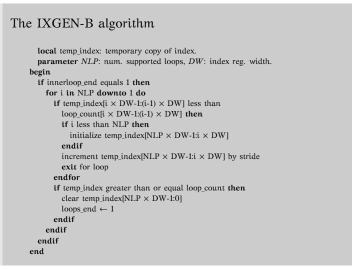

Hardware algorithm(s) for zero-overhead looping on perfect nests

- The purpose of a hardware algorithm is to automate the design of compact and efficient hardware looping units that can be implemented as fully synchronous hardware
- HWLUs are kind of “tuple generators” covering the space of d-tuples for d-dimensional data processing
- There are two forms of the basic generation algorithm
  - **IXGEN-B**: describes a parameterized HDL model for any number of loops
  - **IXGEN-R**: describes a VHDL code generator of an equivalent index generation unit. It uses a priority encoded scheme that cannot be specified in a parameterized manner using natural HDL semantics

- No additional comments

The IXGEN-B algorithm

```
    local temp_index: temporary copy of index.
    parameter NLP: num. supported loops, DW: index reg. width.
  begin
    if innerloop_end equals 1 then
      for i in NLP downto 1 do
        if temp_index[i × DW-1:(i-1) × DW] less than
          loop_count[i × DW-1:(i-1) × DW] then
          if i less than NLP then
            initialize temp_index[NLP × DW-1:i × DW]
          endif
          increment temp_index[NLP × DW-1:i × DW] by stride
          exit for loop
        endfor
        if temp_index greater than or equal loop_count then
          clear temp_index[NLP × DW-1:0]
          loops_end ← 1
        endif
      endif
    endif
  end
```



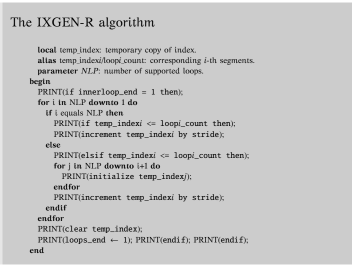
The IXGEN-B algorithm

```
    local temp_index: temporary copy of index.
    parameter NLP: num. supported loops, DW: index reg. width.
  begin
    if innerloop_end equals 1 then
      for i in NLP downto 1 do
        if temp_index[i × DW-1:(i-1) × DW] less than
          loop_count[i × DW-1:(i-1) × DW] then
          if i less than NLP then
            initialize temp_index[NLP × DW-1:i × DW]
          endif
          increment temp_index[NLP × DW-1:i × DW] by stride
          exit for loop
        endfor
        if temp_index greater than or equal loop_count then
          clear temp_index[NLP × DW-1:0]
          loops_end ← 1
        endif
      endif
    endif
  end
```

- *IXGEN-B* produces a behavioral VHDL model for any number of loops
- *loop_count* and *index* are vectorized forms of the set of loop bound values and the current iteration vector, correspondingly
- When the data processing in the inner loop is completed, *innerloop_end* is asserted and a cascaded set of comparisons between index registers to their corresponding loop bound values is activated
- The flow of comparisons is directed from outermost to their immediately innermost loops
- If the index value is less than the loop bound for a given loop *i*, the index is incremented by a stride value, while all its outer loops are set to the initial index values
- After the first successful comparison, the cascaded structure is exited by a `break`-like condition mechanism

The IXGEN-R algorithm

```
    local temp_index: temporary copy of index.
    alias temp_indexi/loopi_count: corresponding i-th segments.
    parameter NLP: number of supported loops.
begin
    PRINT(if innerloop_end = 1 then);
    for i in NLP downto 1 do
      if i equals NLP then
        PRINT(if temp_indexi <= loopi_count then);
        PRINT(increment temp_indexi by stride);
      else
        PRINT(elsif temp_indexi <= loopi_count then);
        for j in NLP downto i+1 do
          PRINT(initialize temp_indexj);
        endfor
        PRINT(increment temp_indexi by stride);
      endif
    endfor
    PRINT(clear temp_index);
    PRINT(loops_end ← 1); PRINT(endif); PRINT(endif);
end
```

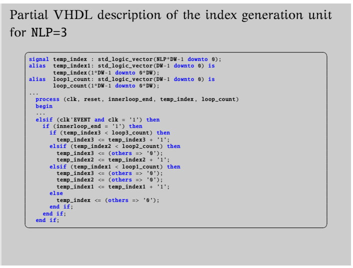

The IXGEN-R algorithm

```
    local temp_index: temporary copy of index.
    alias temp_indexi/loopi_count: corresponding i-th segments.
    parameter NLP: number of supported loops.
begin
    PRINT(if innerloop_end = 1 then);
    for i in NLP downto 1 do
      if i equals NLP then
        PRINT(if temp_indexi <= loopi_count then);
        PRINT(increment temp_indexi by stride);
      else
        PRINT(elsif temp_indexi <= loopi_count then);
        for j in NLP downto i+1 do
          PRINT(initialize temp_indexj);
        endfor
        PRINT(increment temp_indexi by stride);
      endif
    endfor
    PRINT(clear temp_index);
    PRINT(loops_end ← 1); PRINT(endif); PRINT(endif);
end
```

- *IXGEN-R* describes an HDL code generator of an equivalent index generation unit at the register transfer level
- The main difference to *IXGEN-B* is that it has been adapted to the generation of RTL designs with a hard-coded priority encoding scheme
- The temporary signals *tempn_index* and *loop_countn* are used, where *n* is the current loop enumeration
- All lines featuring a call to the `PRINT` routine illustrate emitted code

Partial VHDL description of the index generation unit for NLP=3

```vhdl
signal temp_index : std_logic_vector(NLP*DW-1 downto 0);
alias  temp_index1: std_logic_vector(DW-1 downto 0) is
       temp_index(1*DW-1 downto 0*DW);
alias  loop1_count: std_logic_vector(DW-1 downto 0) is
       loop_count(1*DW-1 downto 0*DW);
...
  process (clk, reset, innerloop_end, temp_index, loop_count)
  begin
  ...
  elsif (clk'EVENT and clk = '1') then
    if (innerloop_end = '1') then
      if (temp_index3 < loop3_count) then
        temp_index3 <= temp_index3 + '1';
      elsif (temp_index2 < loop2_count) then
        temp_index3 <= (others => '0');
        temp_index2 <= temp_index2 + '1';
      elsif (temp_index1 < loop1_count) then
        temp_index3 <= (others => '0');
        temp_index2 <= (others => '0');
        temp_index1 <= temp_index1 + '1';
      else
        temp_index <= (others => '0');
      end if;
    end if;
  end if;
```


Partial VHDL description of the index generation unit for NLP=3

```vhdl
signal temp_index : std_logic_vector(NLP*DW-1 downto 0);
alias  temp_index1: std_logic_vector(DW-1 downto 0) is
       temp_index(1*DW-1 downto 0*DW);
alias  loop1_count: std_logic_vector(DW-1 downto 0) is
       loop_count(1*DW-1 downto 0*DW);
...
  process (clk, reset, innerloop_end, temp_index, loop_count)
  begin
  ...
  elsif (clk'EVENT and clk = '1') then
    if (innerloop_end = '1') then
      if (temp_index3 < loop3_count) then
        temp_index3 <= temp_index3 + '1';
      elsif (temp_index2 < loop2_count) then
        temp_index3 <= (others => '0');
        temp_index2 <= temp_index2 + '1';
      elsif (temp_index1 < loop1_count) then
        temp_index3 <= (others => '0');
        temp_index2 <= (others => '0');
        temp_index1 <= temp_index1 + '1';
      else
        temp_index <= (others => '0');
      end if;
    end if;
  end if;
```

- An example of an index generator of a triple perfect loop nest generated by *IXGEN-R*
- All index values are assumed to be initialized to zero
- The generator produces VHDL'93-compliant code, only partially shown here

Use case 1: Scanning integer points in polyhedra

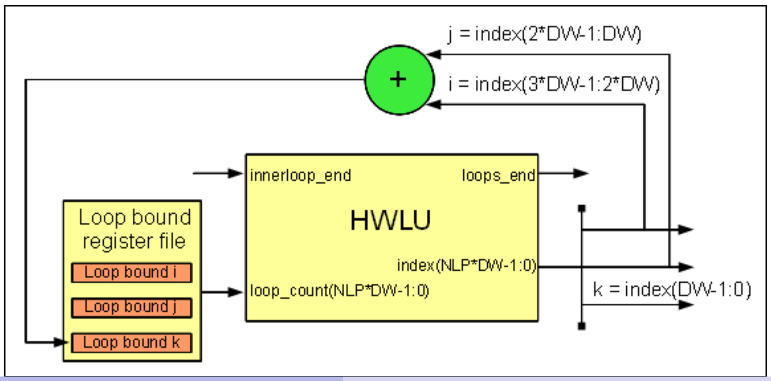
- Assume the 3D polyhedron defined by the inequalities:

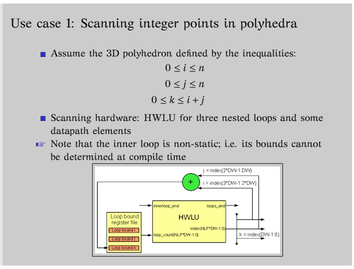
$$0 \le i \le n$$

$$0 \le j \le n$$

$$0 \le k \le i + j$$

- Scanning hardware: HWLU for three nested loops and some datapath elements
- ☞ Note that the inner loop is non-static; i.e. its bounds cannot be determined at compile time





- Consider this three-dimensional polyhedron
- The corresponding implementation of a scanning routine either in software or in hardware would have to visit all the integer points that define the polyhedron
- The upper bound for the inner loop is not static since it depends on the value of indices i, j
- The HWLU serves as part of the necessary control logic, requiring only limited additions, e.g. an adder for computing the $i + j$ sum
- This approach can be easily extended to more intriguing cases such as unions of polyhedra that are of certain interest in the field of high-level synthesis

Use case 2: Kernel applications with general loop structures (1)

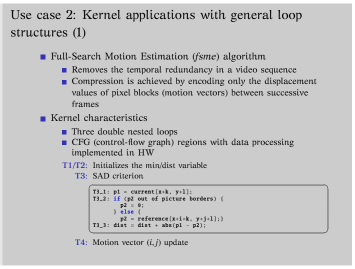
- Full-Search Motion Estimation (*fsme*) algorithm
  - Removes the temporal redundancy in a video sequence
  - Compression is achieved by encoding only the displacement values of pixel blocks (motion vectors) between successive frames
- Kernel characteristics
  - Three double nested loops
  - CFG (control-flow graph) regions with data processing implemented in HW

T1/T2: Initializes the min/dist variable

T3: SAD criterion

```
T3_1: p1 = current[x+k, y+l];
T3_2: if (p2 out of picture borders) {
        p2 = 0;
      } else {
        p2 = reference[x+i+k, y+j+l];}
T3_3: dist = dist + abs(p1 - p2);
```

T4: Motion vector (i, j) update



Use case 2: Kernel applications with general loop structures (1)

- Full-Search Motion Estimation (*fsme*) algorithm
  - Removes the temporal redundancy in a video sequence
  - Compression is achieved by encoding only the displacement values of pixel blocks (motion vectors) between successive frames
- Kernel characteristics
  - Three double nested loops
  - CFG (control-flow graph) regions with data processing implemented in HW

T1/T2: Initializes the min/dist variable
T3: SAD criterion

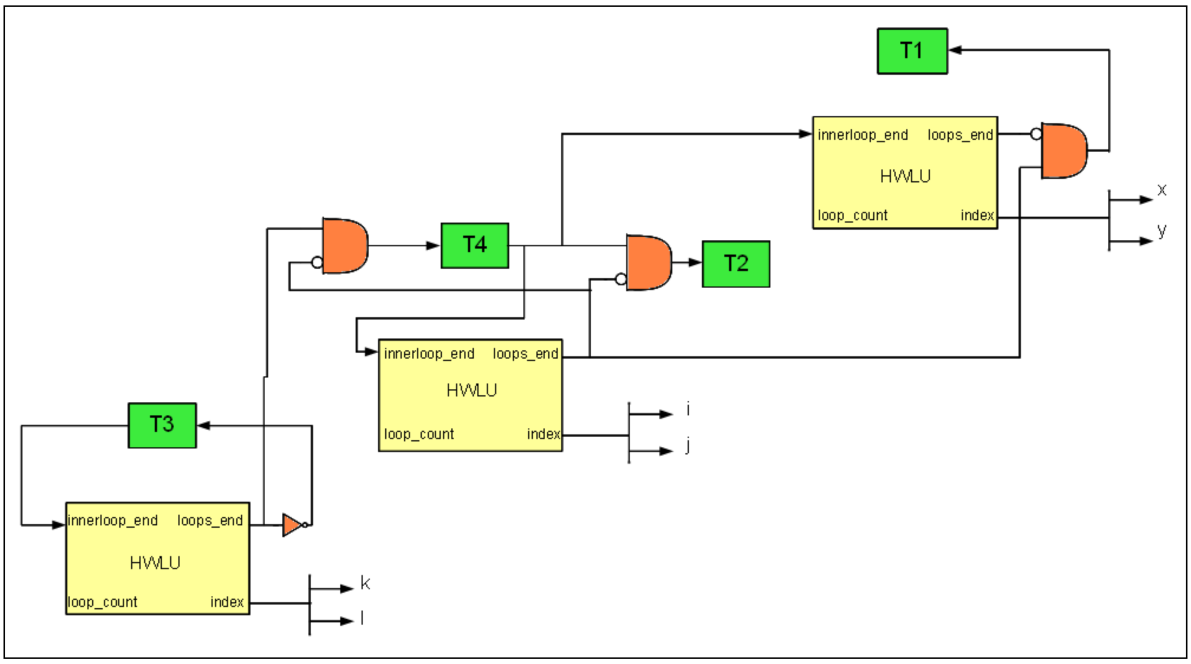
T3_1: p1 = current[x+k, y+l];
T3_2: if (p2 out of picture borders) {
        p2 = 0;
      } else {
        p2 = reference[x+i+k, y+j+l];}
T3_3: dist = dist + abs(p1 - p2);

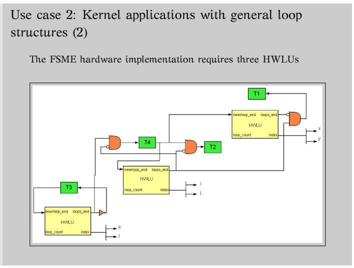
T4: Motion vector (i, j) update

- The HWLU is used for implementing the Full-Search Motion Estimation (*fsme*) algorithm
- The calculation of the motion vector is performed by a cost function minimizing the prediction error
- The *fsme* algorithm consists of three double nested loops incorporating the data processing tasks of the algorithm

Use case 2: Kernel applications with general loop structures (2)

The FSME hardware implementation requires three HWLUs



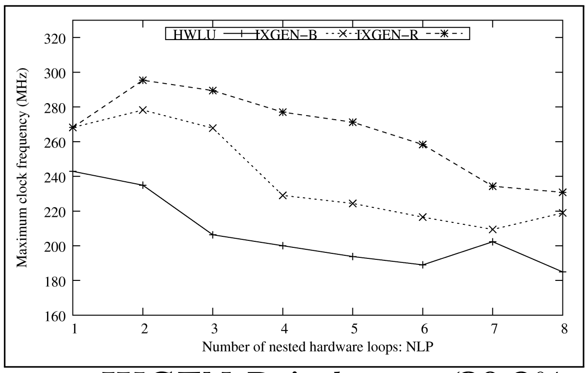


- The *fsme* algorithm consists of three double nested loops incorporating the data processing tasks of the algorithm
- The outer (x, y) loops select the block from the current picture for which the minimum motion vector is calculated
- By iterating (i, j), each time a reference block is selected from the reference window
- For each position in the search region, the distance kernel is executed, and this is performed for all (k, l) pixels in the current picture block
- Each double loop nest is assigned its dedicated HWLU instance
- Updating the iteration vector is enabled by the termination of tasks T3 and T4 which are positioned at a closing position for a loop [Kavvadias:08]

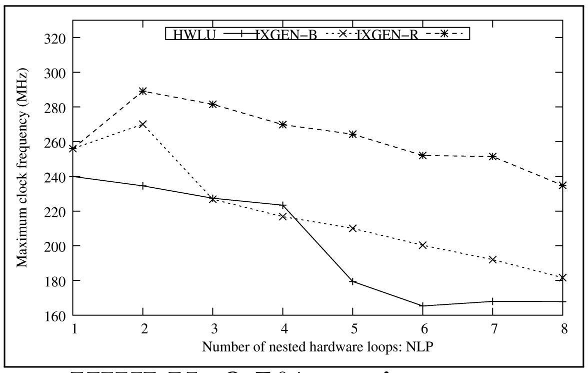
Performance results (speed measurements)

- Three variants are compared: HWLU (hand-optimized VHDL), IXGEN-B (behavioral), IXGEN-R (RTL) have been synthesized on XC5VLX50 (Xilinx Virtex-5)
- Parameter set: $NLP : 1 - 8$ and $DW : 8, 16$ bits

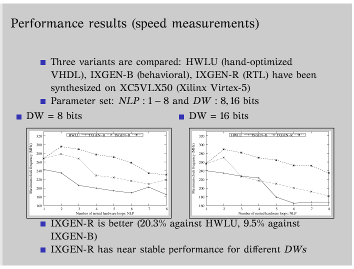
- DW = 8 bits



- DW = 16 bits



- IXGEN-R is better (20.3% against HWLU, 9.5% against IXGEN-B)
- IXGEN-R has near stable performance for different *DWs*

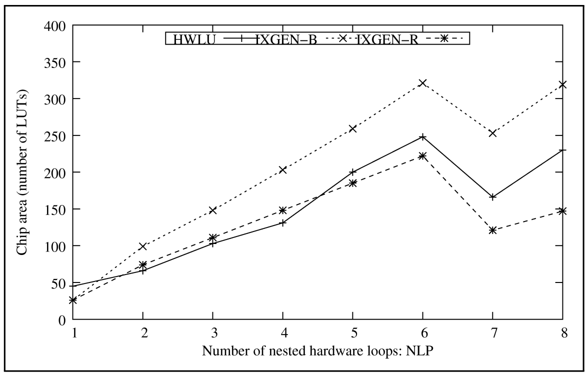


- The figures depict the maximum clock frequency estimates for different number of supported maximum number of loops (*NLP*={1 ... 8}) and for different index register widths (*DW* = 8, 16)
- The IXGEN-R design achieves nearly unvarying performance due to the fact that the synthesis tool efficiently balances the index increment logic for the prioritized cases, the evaluation of which has the same logic depth in an FPGA implementation
- Both the HWLU and the IXGEN-B designs don't scale gracefully with increased values of *DW*, since the synthesis tool infers cascaded logic

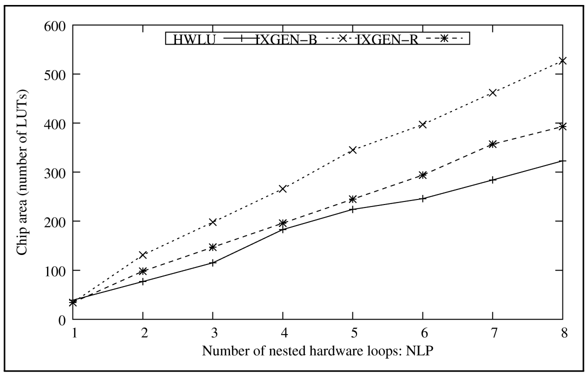
Performance results (chip area measurements)

- For the same parameter set

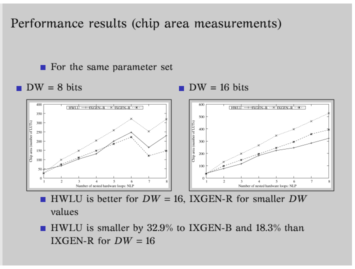
- DW = 8 bits



- DW = 16 bits



- HWLU is better for $DW = 16$, IXGEN-R for smaller DW values
- HWLU is smaller by 32.9% to IXGEN-B and 18.3% than IXGEN-R for $DW = 16$

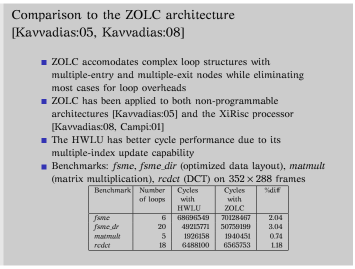

Performance results (chip area measurements)
- For the same parameter set
- DW = 8 bits
- DW = 16 bits
- HWLU is better for $DW = 16$, IXGEN-R for smaller DW values
- HWLU is smaller by 32.9% to IXGEN-B and 18.3% than IXGEN-R for $DW = 16$

- This observation on chip area (HWLU vs IXGEN-R) can be explained by taking account the sparsely populated logic slices in the HWLU design for the small DW values
- Many of these slices get populated when DW is increased and hardware exploitation for HWLU is significantly improved
- On the contrary, the IXGEN-B and IXGEN-R designs feature more compact descriptions that leave no room for such behavior

Comparison to the ZOLC architecture [Kavvadias:05, Kavvadias:08]

- ZOLC accomodates complex loop structures with multiple-entry and multiple-exit nodes while eliminating most cases for loop overheads
- ZOLC has been applied to both non-programmable architectures [Kavvadias:05] and the XiRisc processor [Kavvadias:08, Campi:01]
- The HWLU has better cycle performance due to its multiple-index update capability
- Benchmarks: *fsme*, *fsme_dir* (optimized data layout), *matmult* (matrix multiplication), *rcdct* (DCT) on 352×288 frames

Benchmark	Number of loops	Cycles with HWLU	Cycles with ZOLC	%diff
fsme	6	68696549	70128467	2.04
fsme_dr	20	49215771	50759199	3.04
matmult	5	1926158	1940451	0.74
rcdct	18	6488100	6565753	1.18


Comparison to the ZOLC architecture [Kavvadias:05, Kavvadias:08]

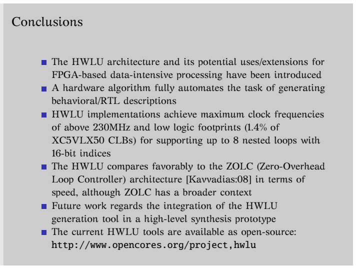
- ZOLC accomodates complex loop structures with multiple-entry and multiple-exit nodes while eliminating most cases for loop overheads
- ZOLC has been applied to both non-programmable architectures [Kavvadias:05] and the XiRisc processor [Kavvadias:08, Campi:01]
- The HWLU has better cycle performance due to its multiple-index update capability
- Benchmarks: *fsme*, *fsme_dir* (optimized data layout), *matmult* (matrix multiplication), *rcdct* (DCT) on 352×288 frames

Benchmark	Number of loops	Cycles with HWLU	Cycles with ZOLC	%diff
fsme	6	68696549	70128467	2.04
fsme_dr	20	49215771	50759199	3.04
matmult	5	1926158	1940451	0.74
rcdct	18	6488100	6565753	1.18

- No additional comments

Conclusions

- The HWLU architecture and its potential uses/extensions for FPGA-based data-intensive processing have been introduced
- A hardware algorithm fully automates the task of generating behavioral/RTL descriptions
- HWLU implementations achieve maximum clock frequencies of above 230MHz and low logic footprints (1.4% of XC5VLX50 CLBs) for supporting up to 8 nested loops with 16-bit indices
- The HWLU compares favorably to the ZOLC (Zero-Overhead Loop Controller) architecture [Kavvadias:08] in terms of speed, although ZOLC has a broader context
- Future work regards the integration of the HWLU generation tool in a high-level synthesis prototype
- The current HWLU tools are available as open-source: `http://www.opencores.org/project,hwlu`

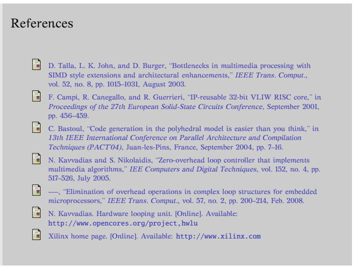

Conclusions

- The HWLU architecture and its potential uses/extensions for FPGA-based data-intensive processing have been introduced
- A hardware algorithm fully automates the task of generating behavioral/RTL descriptions
- HWLU implementations achieve maximum clock frequencies of above 230MHz and low logic footprints (1.4% of XC5VLX50 CLBs) for supporting up to 8 nested loops with 16-bit indices
- The HWLU compares favorably to the ZOLC (Zero-Overhead Loop Controller) architecture [Kavvadias:08] in terms of speed, although ZOLC has a broader context
- Future work regards the integration of the HWLU generation tool in a high-level synthesis prototype
- The current HWLU tools are available as open-source: http://www.opencores.org/project,hwlu

- No additional comments

References

- D. Talla, L. K. John, and D. Burger, “Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements,” *IEEE Trans. Comput.*, vol. 52, no. 8, pp. 1015–1031, August 2003.
- F. Campi, R. Canegallo, and R. Guerrieri, “IP-reusable 32-bit VLIW RISC core,” in *Proceedings of the 27th European Solid-State Circuits Conference*, September 2001, pp. 456–459.
- C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in *13th IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, Juan-les-Pins, France, September 2004, pp. 7–16.
- N. Kavvadias and S. Nikolaidis, “Zero-overhead loop controller that implements multimedia algorithms,” *IEE Computers and Digital Techniques*, vol. 152, no. 4, pp. 517–526, July 2005.
- ——, “Elimination of overhead operations in complex loop structures for embedded microprocessors,” *IEEE Trans. Comput.*, vol. 57, no. 2, pp. 200–214, Feb. 2008.
- N. Kavvadias. Hardware looping unit. [Online]. Available: `http://www.opencores.org/project,hwlu`
- Xilinx home page. [Online]. Available: `http://www.xilinx.com`


References

D. Talla, L. K. John, and D. Burger, “Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements,” *IEEE Trans. Comput.*, vol. 52, no. 8, pp. 1015–1031, August 2003.

F. Campi, R. Canegallo, and R. Guerrieri, “IP-reusable 32-bit VLIW RISC core,” in *Proceedings of the 27th European Solid-State Circuits Conference*, September 2001, pp. 456–459.

C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in *13th IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, Juan-les-Pins, France, September 2004, pp. 7–16.

N. Kavvadias and S. Nikolaidis, “Zero-overhead loop controller that implements multimedia algorithms,” *IEE Computers and Digital Techniques*, vol. 152, no. 4, pp. 517–526, July 2005.

——, “Elimination of overhead operations in complex loop structures for embedded microprocessors,” *IEEE Trans. Comput.*, vol. 57, no. 2, pp. 200–214, Feb. 2008.

N. Kavvadias. Hardware looping unit. [Online]. Available: `http://www.opencores.org/project,hwlu`

Xilinx home page. [Online]. Available: `http://www.xilinx.com`

- No additional comments