

# Generation of optimized DSP library for mAgicV VLIW DSP

**How to write optimized code for the mAgicV VLIW DSP**

**Elena Pastorelli**

Atmel Roma

[elena.pastorelli@atmelroma.it](mailto:elena.pastorelli@atmelroma.it)

**CASTNESS'07**



## Agenda

- **mAgic Instruction Level Parallelism**
- **DSP Optimized Library**
- **Main Optimization Techniques for mAgicV**
  - **High Level Code Optimizations**
  - **Dedicated Assembler Optimizations**
- **Examples**
- **Conclusions**



## mAgicV Instruction Level Parallelism

- **mAgicV DSP is a Very Long Instruction Word**
  - The impressive internal data bandwidth supports 5 VLIW Issues
- **All the instructions are pipelined**
  - The different devices involved in each instruction are activated at the proper stage

Instruction Pipelining



V L I W				
FLOW	AGU0	MUL	AGU1	ADD

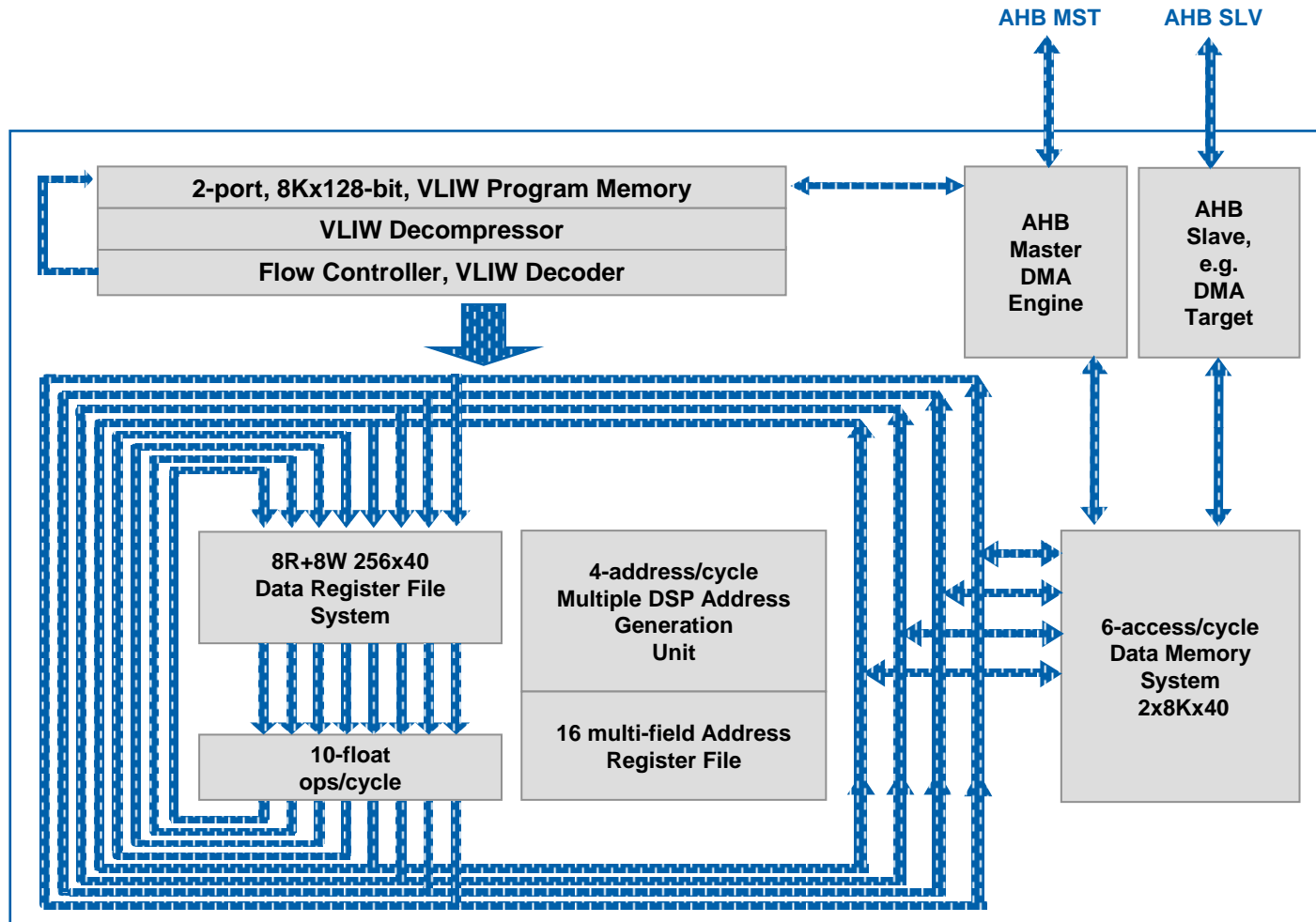
The code activates  
the right ISSUE...  
...at the right time



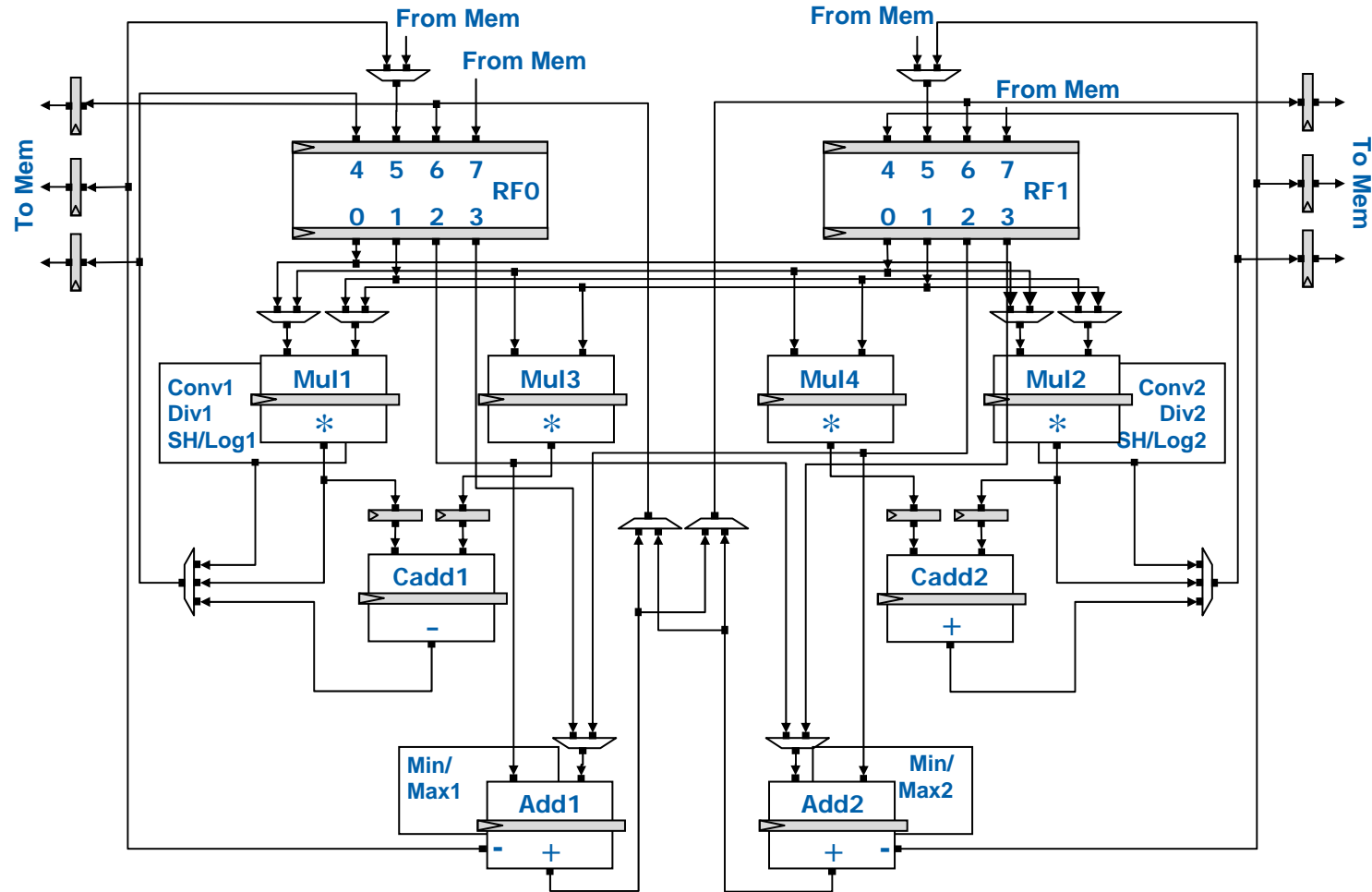
## mAgic ILP Example

- **Example of what can be executed in the same cycle in the mAgicV DSP:**
  - **10 floating point operation**
    - 16 \* 40-bit data read/written on the multiport Data Register File
  - **4 memory accesses**
    - 8 \* 16-bit address fields read/written on the multiport Address Register File
    - 2 addresses update
  - **1 flow control instruction**
  - **1 DMA access**
  - **1 AHB access (managed by ARM)**

# mAgicV Architecture



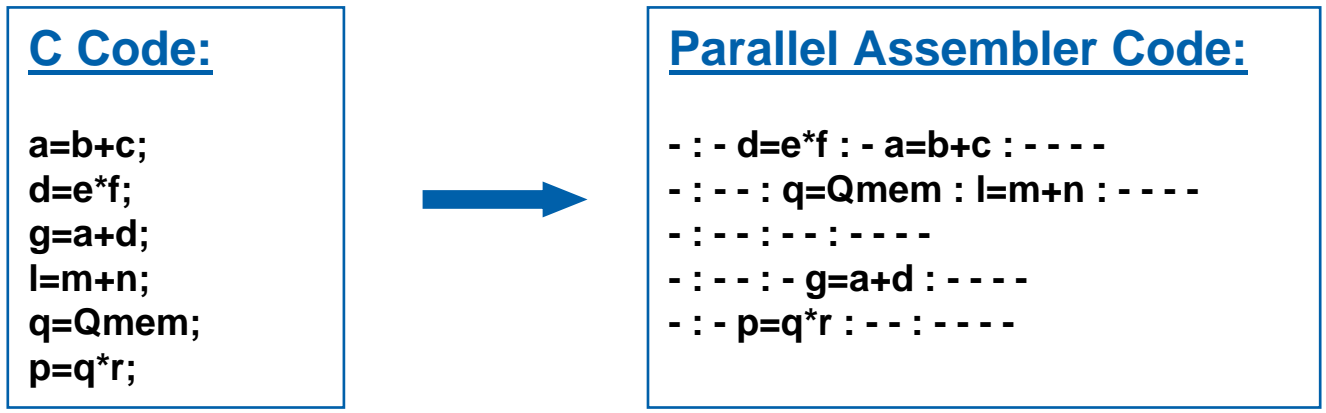
# mAgicV Operators Block





## C Code vs. mAgicV Parallel Assembler

- The mAgicV C compiler act as a scheduler optimizer, producing a parallel assembler that takes advantage of the DSP Instruction Level Parallelism, accounting for data dependencies and latencies



- The maximum parallelism level is achieved → Faster&shorter code
- The order of the instructions can be inverted
- Dependencies between instructions are always maintained



## DSP Library

- **C callable optimized functions performing computations of some algorithms typical of DSP applications**
- **All the functions works on array of the following types:**
  - float / long
  - `_v_float` / `_v_long`
  - `_c_float` / `_c_long`
- **Main groups of functions:**
  - **Simple:** array addition, fill, move, mul, fix, clip, sum...
  - **Trigonometric and hyperbolic:** sin, asin, sinh, asinh...
  - **Power:** log, exp
  - **Matrix:** add, mul, determ, inverse, decomposition, trace...
  - **Miscellaneous:** sort, rand, sqrt, div, max, dist...
  - **DSP:** Cross-Correlation, Convolution
  - **Filters:** different implementations of FIRs and IIRs
  - **FFT and iFFT:** FFTs and iFFT for several number of points (1024, 512, 256...)

## DSP Library Generation

- **The library has been generated using the mAgicV C Compiler**
- **Only inner kernels have been optimized in C using the optimization techniques here described**
- **No optimization at parallel assembler level have been necessary**
- **Performances lies among 80% - 100% of theoretical cycle consumption estimated for the mAgicV DSP**

## Efficient Optimizations

- **Main techniques for an optimized code on the mAgic VLIW DSP:**
  - **Memory Disambiguation**
  - **Register Dependencies Elimination**
  - **Loop Unrolling**
  - **Software Pipelining**
  - **Loop Count Annotation**
  - **Instruction Predication (only assembler)**
  - **HW Support to SW-Pipelining (only assembler)**

## Memory Disambiguation

- **The compiler is able to get the best schedule when the pointers involved in the computation points to independent memory areas**
  - **It is able to freely move the writing and the reading accesses insisting on different memory areas, searching for the best optimization**
    - ➔ **Reduce memory dependencies instructing the compiler about independent pointers**
    - ➔ **Use the “restrict” qualifier for the pointers addressing independent memory areas**



## Register Dependencies Elimination

- **Using an higher number of independent registers, the code can be easily compacted by the compiler**
  - **The use of independent registers wherever is possible brings to the elimination of dependencies between instructions**
    - ➔ **Reduce register dependencies using the project option -b for the Showcolor module (compiler default option)**

# Register Dependencies Elimination Example

## C code

```

for(i=0;i<128;i++)
{
    data_a = input1[i];
    data_b = input2[i];
    data_d = input3[i];
    data_e = input4[i];

    data_a = data_a * data_b;
    data_c = data_a + data_b;
    data_e = data_d * data_e;
    data_f = data_d + data_e;

    output1[i] = data_c;
    output2[i] = data_f;
}
    
```

Compiler forced to use only 3 registers



## Parallel code

```

- : RF0x14 = data_a : - : RF0x16 = data_b : - : - : - : - : - :
- : - : - : - : - : - : - :
- : - : - : - : - : - : - :
- : - : RF0x14 = RF0x14 * RF0x16 : - : - : - : - : - :
- : - : RF0x16 = data_e : - : - : RF0x14 = data_d : - : - : - : - :
- : - : - : - : - : - : - :
- : - : - : RF0x18 = RF0x14 + RF0x16 : - : - : - : - : - :
- : - : RF0x16 = RF0x14 * RF0x16 : - : - : - : - : - :
- : - : - : - : - : - : - :
- : - : - : data_c = RF0x18 : - : - : - : - : - :
- : - : - : RF0x18 = RF0x14 + RF0x16 : - : - : - : - : - :
- : - : - : - : - : - : - :
- : - : - : - : - : - : - :
- : - : - : data_f = RF0x18 : - : - : - : - : - :
    
```

Compiler free to use all the registers



## Parallel code

```

- : - : RF0xe = data_a : - : - : RF0x10 = data_b : - : - : - : - : - :
- : - : RF0x14 = data_d : - : - : RF0x12 = data_e : - : - : - : - : - :
- : - : - : - : - : - : - :
- : - : - : RF0x16 = RF0xe * RF0x10 : - : - : - : - : - :
- : - : - : RF0x18 = RF0x14 * RF0x12 : - : - : - : - : - :
- : - : - : - : - : - : - :
- : - : - : data_c = RF0xe + RF0x16 : - : - : - : - : - :
- : - : - : data_f = RF0x18 + RF0x12 : - : - : - : - : - :
    
```

## Loop Unrolling

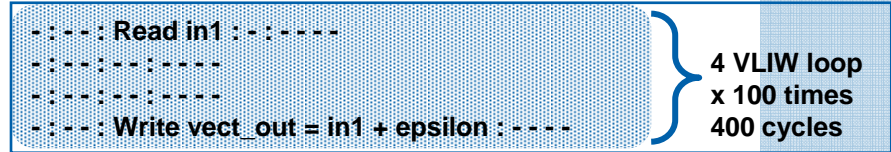
- **Branches constitute a cut in the code**
  - The compiler can't perform any kind of optimization across them
  - The instruction pipeline must be emptied before crossing this cut (even if, wherever is possible, the tail of the loop is closed on the beginning of the same loop, without waiting the end of the operation)
  
- **Totally unrolling the loop, branches are avoided**
  - Time spent in branch initialization and in branch execution is saved
  - The code can be better optimized
  
- **Loop unrolling have to be of the correct size**
  - Unrolling large loops, PM occupation grows enormously
  - The correct size is the one that allows to fill the operators pipeline (typically 4)
  - In association with other optimization techniques (above all sw-pipelining) and in loops dominated by computation, the unroll can be reduced to 2 or 1
  
- **The loop unroll can be:**
  - **Manual:** the user write the code duplicating the instructions inside the loop
  - **Automatic:** using the pragma `chess_unroll_loop(n)`

# Loop Unrolling Example

## C code without Loop Unrolling

```
for (i=0;i<100;i++)
{
    vect_out[i] = in1[i] + epsilon ;
}
```

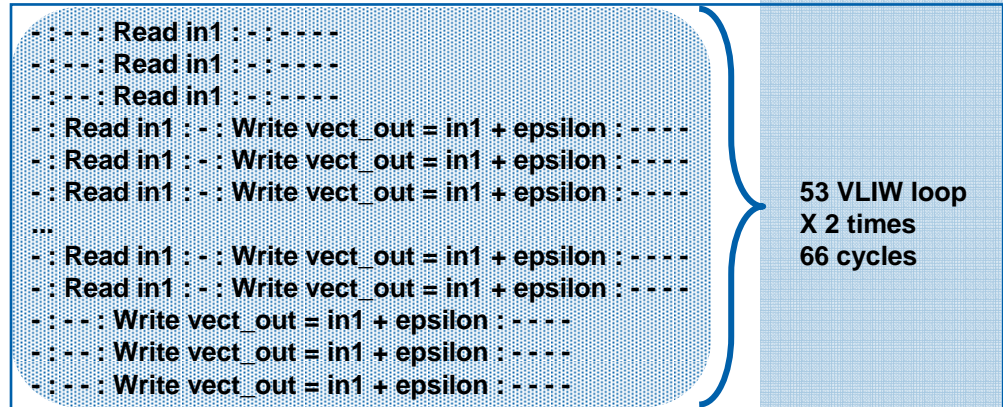
## Parallel code



## C code with Loop Unrolling

```
for (i=0;i<100;i++) chess_unroll_loop(50)
{
    vect_out[i] = in1[i] + epsilon ;
}
```

## Parallel code



## Software Pipelining

- **Software pipelining is probably the most important code optimization for mAgicV computational kernels**
  
- **Software pipelining can be:**
  - **Automatic: enabled by default**
  - **Manual: the user writes the C instructions of the loop in the appropriate way**
  - **Both automatic and manual (needed only for more complex loops)**
  
- **Usually the software pipelining automatically done by the compiler is sufficient for very good performances**



## Software Pipelining Example (1/3)

- Starting from the following linear code, the corresponding assembler parallel code contains nearly no parallelization
- Execution cycles:  $4 + 7 \times 64 = 452$
- Code length: 4 (initialization) + 7 (loop) = 11 VLIWs

### Linear code

```

Acc = 0

for (i=0; i<64; i++)
{
  Read X
  Read H
  Mul = X * H
  Acc = Acc + Mul
}
    
```



### Parallel code

```

REPEAT ;-;-;-;-;- (loop instruction)
Acc = 0 ;-;-;-;-;- (nop)
;-;-;-;-;- (nop)
;-;-;-;-;- (nop)
;-; Read X ; -; Read H ; -; -;-;- (Read X || Read H)
;-;-;-;-;- (nop)
;-;-;-;-;- (nop)
;-;-; Mul = X * H ; -;-;-;-;- (Mul)
;-;-;-;-;- (nop)
;-;-;-;-;- (nop)
;-;-;-;-; Acc = Acc + Mul ; -;-;-;- (Add)
    
```

} loop x 64 times





## Software Pipelining Example (3/3)

- Further optimization: for code size reduction, the epilogue can be avoided, taking care of masking possible arithmetic exceptions
- Execution cycles:  $6 + 4 \times 64 = 262$
- Code length: 6 (prologue) + 4 (loop) = 10 VLIWs

### SW-pipelined code without epilogue

```

Acc = 0
Read X
Read H
Mul = X * H
Read X
Read H
for (i=0; i<64; i++) {
    Acc = Acc + Mul
    Mul = X * H
    Read X
    Read H }
    
```



### Parallel code

```

- ; Read X ; - ; Read H ; - ; ----
- ; - ; - ; - ; - ; ----
REPEAT ; - ; - ; - ; - ; ----
Acc = 0 ; Read X ; Mul = X * H ; Read H ; - ; ----
- ; - ; - ; - ; - ; ----
- ; - ; - ; - ; - ; ----
- ; Read X ; Mul = X * H ; Read H ; Acc = Acc + Mul ; - ; ----
- ; - ; - ; - ; - ; ----
- ; - ; - ; - ; - ; ----
- ; - ; - ; - ; - ; ----
    
```

} loop  
x 64 times





# SW-Pipelining & Loop Unrolling Example

- For further optimization is sometimes useful combining the Loop Unrolling and the SW-Pipelining techniques
- Execution cycles:  $8 + 4 \times 16 = 72$
- Code length: 8 (prologue) + 4 (loop) = 12 VLIWs

## SW-pipelined code with unroll 4

```

.... (prologue)
for (i=0; i<16; i++)
{
    Acc0 = Acc0 + Mul0
    Acc1 = Acc1 + Mul1
    Acc2 = Acc2 + Mul2
    Acc3 = Acc3 + Mul3
    Mul0 = X0 * H0
    Mul1 = X1 * H1
    Mul2 = X2 * H2
    Mul3 = X3 * H3
    Read X0, X1, X2, X3
    Read H0, H1, H2, H3
}
    
```



## Parallel code

```

Acc0 = 0 ; Read X0 ; - ; Read H0 ; - ; ----
Acc1 = 0 ; Read X1 ; - ; Read H1 ; - ; ----
Acc2 = 0 ; Read X2 ; - ; Read H2 ; - ; ----
Acc3 = 0 ; Read X3 ; - ; Read H3 ; - ; ----
- ; Read X0 ; Mul0 = X0 * H0 ; Read H0 ; - ; ----
REPEAT ; Read X1 ; Mul1 = X1 * H1 ; Read H1 ; - ; ----
- ; Read X2 ; Mul2 = X2 * H2 ; Read H2 ; - ; ----
- ; Read X3 ; Mul3 = X3 * H3 ; Read H3 ; - ; ----
- ; Read X0 ; Mul0 = X0 * H0 ; Read H0 ; Acc0 = Acc0 + Mul0 ; ----
- ; Read X1 ; Mul1 = X1 * H1 ; Read H1 ; Acc1 = Acc1 + Mul1 ; ----
- ; Read X2 ; Mul2 = X2 * H2 ; Read H2 ; Acc2 = Acc2 + Mul2 ; ----
- ; Read X3 ; Mul3 = X3 * H3 ; Read H3 ; Acc3 = Acc3 + Mul3 ; ----
    
```

} loop  
x 16 times



## Loop Count Annotation

- **Used when the loop count is not known at compilation time, but must be derived from the C code execution**
  
- **The mAgic C compiler can be informed about the minimum number of times the loop will be executed**
  - **This will avoid initial tests on the computed loop counter and code dedicated to particular number of iterations (0, 1 or also 2) that could be no compatible with compiler optimizations**
    - **Use the `chess_loop_range` pragma**
  
    - **Code optimized in size and speed**

## Instruction Predication

- In mAgicV **assembler**, each instruction can be predicated using one of the four available predication registers, previously set with the result of a compare instruction
  - The predicated instructions are executed, but the results are written only if the predication register is “true”
- The use of predication allows to minimize the use of branches, in order to increase the scheduler performances
  - The predicated instructions can be scheduled in parallel with:
    - not-predicated instructions
    - instructions predicated with different predication registers



## HW Support to the SW Pipelining

- **mAgicV assembler** provides a support for the implementation of loop with software pipelines
  - Thanks to a hardware mechanism, prologues and epilogues can be avoided
  - The instructions contained in the prologue and in the epilogue are executed reading the code directly from the loop
  
- **Gain in code size**
  - All the code is contained inside the loop
  
- **Lost in performances**
  - The whole loop is executed for some extra iteration in order to execute the instructions belonging to prologue and to epilogue

## Conclusions

- **Recipe to get an optimized code:**
  - **Write the code without any optimization, using calls to the DSP library functions, if necessary**
  - **Use annotations and pragmas for automatic optimizations (restrict for pointers, chess\_loop\_range,...)**
  - **Analyze the compiler output**
  - **If the optimization is not sufficient, try to add software pipelining by hand**
  - **Add loop unrolling if necessary, manual or automatic**
  - **Reiterate on the last two points until the performances are reached**



# Thank You !

