POLITECNICO DI MILANO

# Synthesis and Optimization of Complex Memory Operations

*Tutorial @ FPT Conference 2017 –Melbourne– Australia*

**Christian Pilato**

Università della Svizzera italiana (USI)
Facoltà di Informatica
christian.pilato@usi.ch

**Fabrizio Ferrandi**

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
fabrizio.ferrandi@polimi.it

## Why Memory is so Important?

- Algorithms operate on data and data must be stored somewhere

- Memories are responsible for **70-90% of total cost**
  - In FPGA:
    - Small/frequently-accessed sets of data in distributed registers (low latency, high resource cost)
    - Medium sets of data in BRAMs (limited number of ports, limited number of resources)
    - Large sets of data in the off-chip memory (e.g., DRAM)

**How to efficiently implement all memory operations?**

## Gap between Software and Hardware

- **Software**:
  - *Stack* for storing data determined statically
  - *Heap* for dynamically allocated data
  - *Cache hierarchies* for hiding the latency
  - *Sequential memory operations*

- **Hardware**
  - *Heterogeneous and distributed memories*
  - *Abundant hardware parallelism*
  - *No "flexibility"* during the execution

## Example

registers

```
int p, n;
struct { int a; int b; } in;
int t[256];
…
if(…)
  p = &in.a;
else
  p = &in.b;
…
t[n] = *p;
*p = t[n + 1];
…
```

memories

## Software Semantics

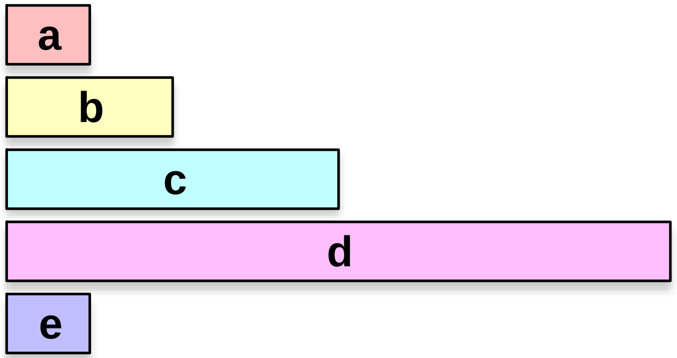
- **Arrays of primitive data types** easily implemented with HDL templates (supported by any synthesis tool)

- There are three types of memory operations to be supported:
  - Accesses to **complex data structures** (e.g., arrays of complex structures)
  - **Parameter passing** in function calls
  - Operation on **pointers** (dynamic resolution, pointer arithmetic, dynamic memory allocation)

**It is relatively easy to implement memory operations, it is NOT easy to implement them efficiently**

## Complex Data Structures

```
typedef struct
{
    char a;         // 1 byte
    short b;        // 2 bytes
    long c;         // 4 bytes
    long long d;    // 8 bytes
    char e;         // 1 byte
} mystruct;     // tot: 16 bytes
```



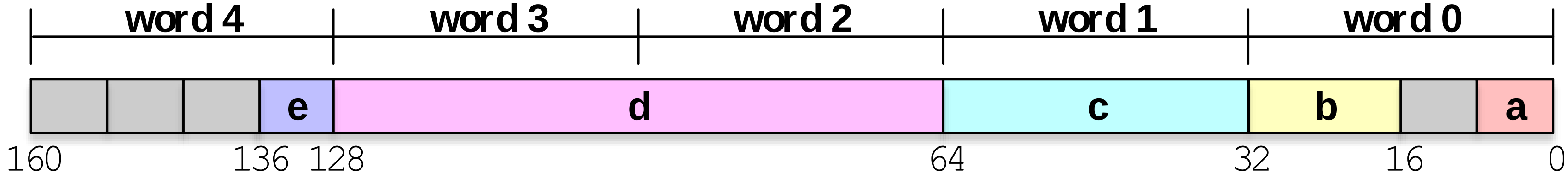
- How to *efficiently* store an array of **`mystruct`** structures?
- How to *efficiently* implement memory operations?

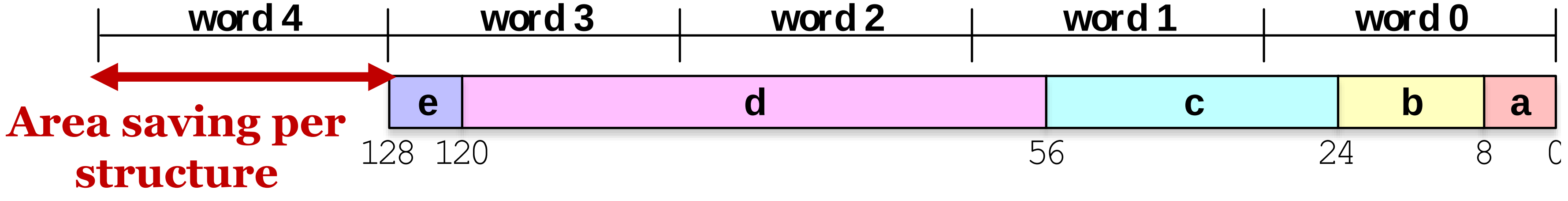
**Reduce the memory footprint without compromising the execution latency**

# Aligned vs. Unaligned Accesses

- Software annotations to create compact representations

```
typedef struct
{
   char a;        // 1 byte
   short b;       // 2 bytes
   long c;        // 4 bytes
   long long d;   // 8 bytes
   char e;        // 1 byte
} __attribute__((packed)) mystruct;
```





## Customizable Hardware Template

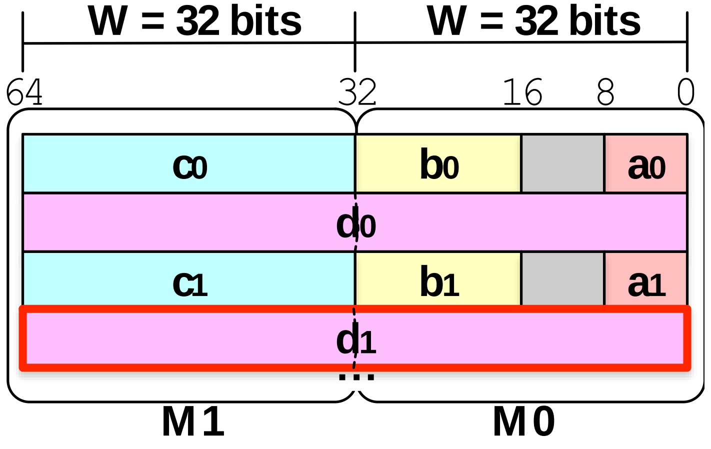
- Alternative solutions:
  - Vivado HLS decomposes the *array of structures* into *multiple arrays of single fields*
  - LegUp uses *subarrays* as large as the maximum field (support only for aligned accesses)

- We use a **single memory template** to implement both aligned and unaligned accesses
  - *Portable* and *easy to maintain*
  - *Simple logic* to convert datapath requests into actual memory operations

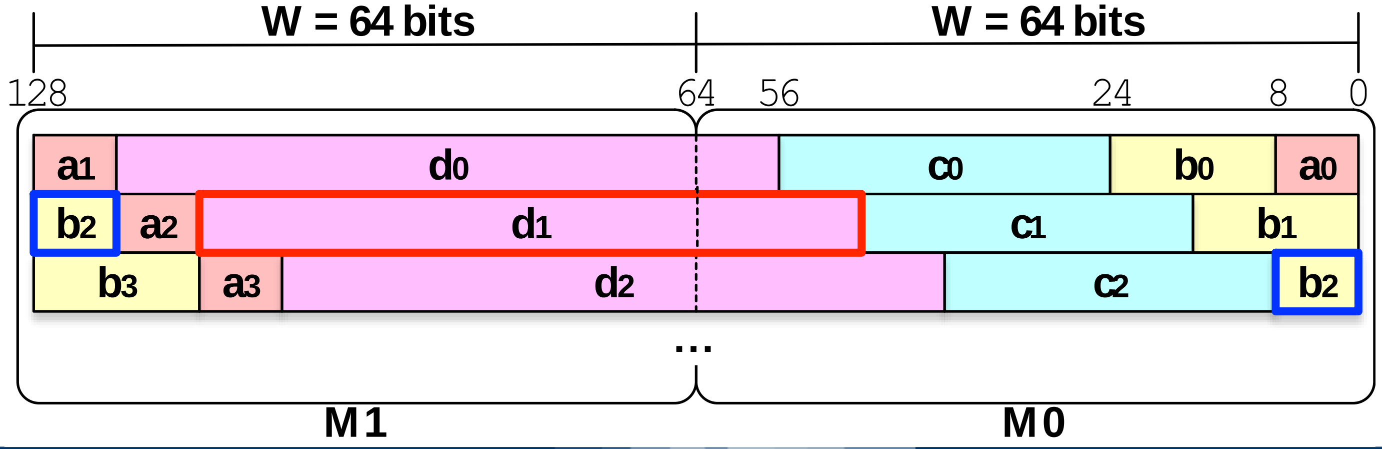
## Memory Layouts

- Two arrays with bitwidth based of the type of accesses
  - At most two (parallel) memory ops for each field

**Aligned structures**
(W = SMAX/2)



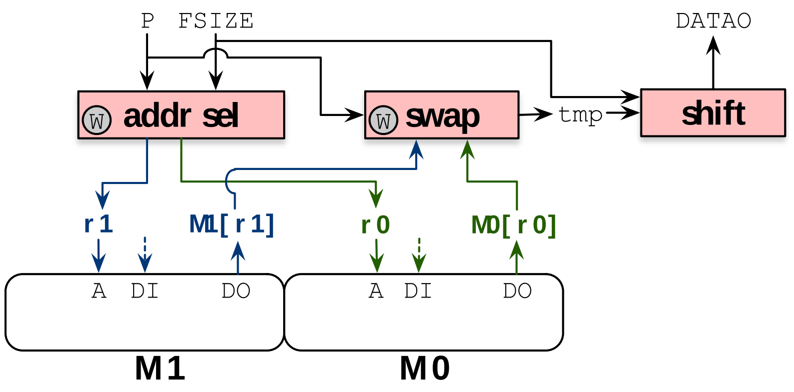
**Unaligned structures**
(W = SMAX)



## Read and Write Operations

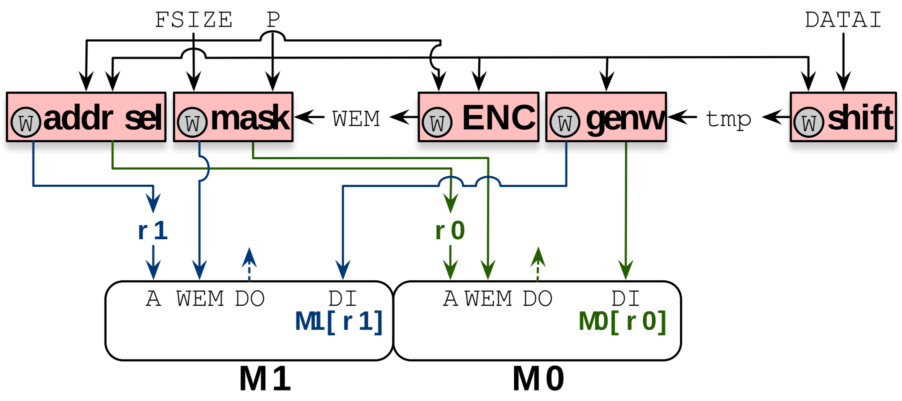
- **Unique microarchitecture** for both aligned and unaligned accesses

**Read operations**



```
array[0].a -> M0[0][7:0]
array[0].b -> M0[0][23:8]
array[0].c -> M0[0][55:24]
array[0].d -> {M1[0][55:0],M0[0][63:56]}
...
array[1].d -> {M1[1][47:0],M0[1][63:48]}
...
array[2].b -> {M0[2][7:0],M1[1][63:56]}
```

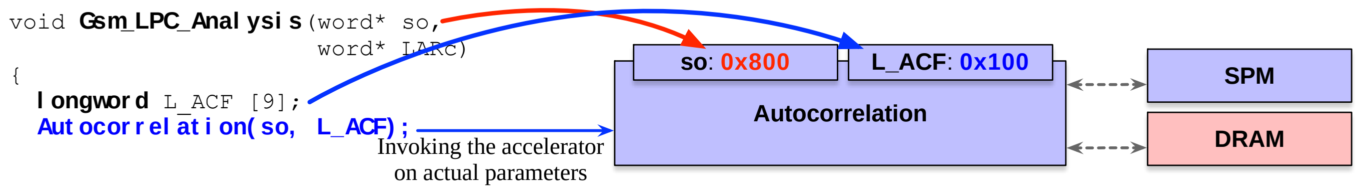
**Write operations**



```
array[0].a -> M0[0][7:0]
array[0].b -> M0[0][31:16]
array[0].c -> M1[0][31:0]
array[0].d -> {M1[1][31:0],M0[1][31:0]}
array[1].a -> M0[2][7:0]
...
array[1].d -> {M1[3][31:0],M0[3][31:0]}
```

## Function Parameters and Calls

- Hardware modules are created hierarchically based on the *call graph*
  - *Software:* parameters on the stack
  - *Hardware:* actual values provided at the input ports



```
void Gsm_LPC_Analysis(word* so,
                      word* LARc)
{
  longword L_ACF [9];
  Autocorrelation(so, L_ACF);
```

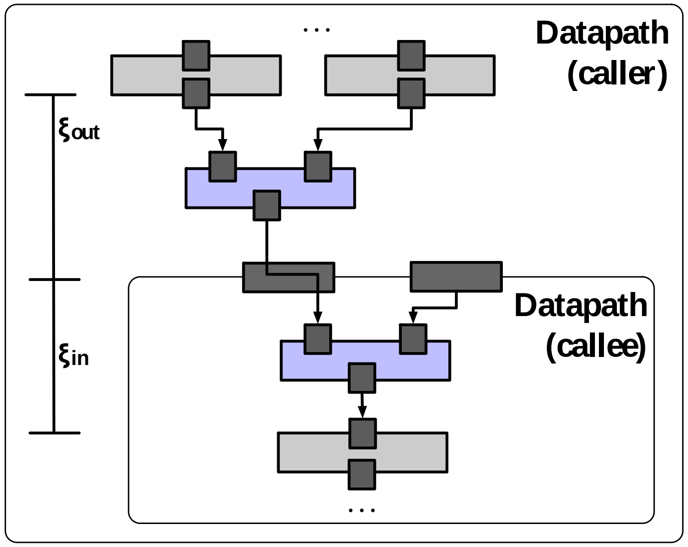
**When to insert registers to avoid long critical paths?**

- We introduce input registers when

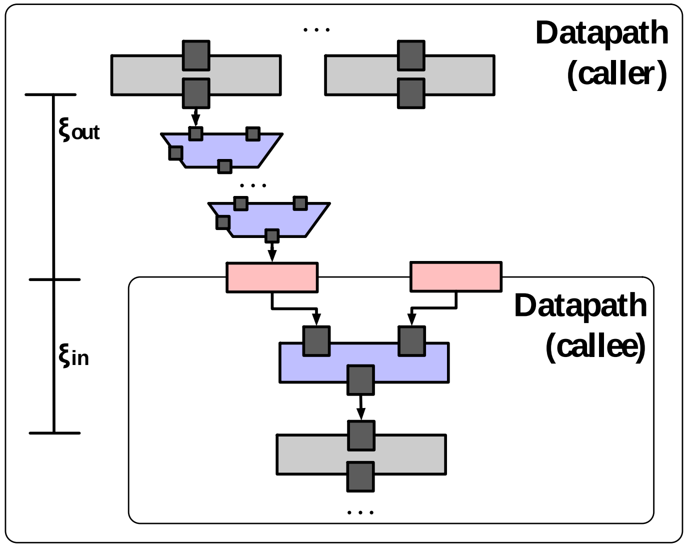
$$(\xi_{out} + \xi_{in}) > T * \beta$$

# Some Examples of Input Registering

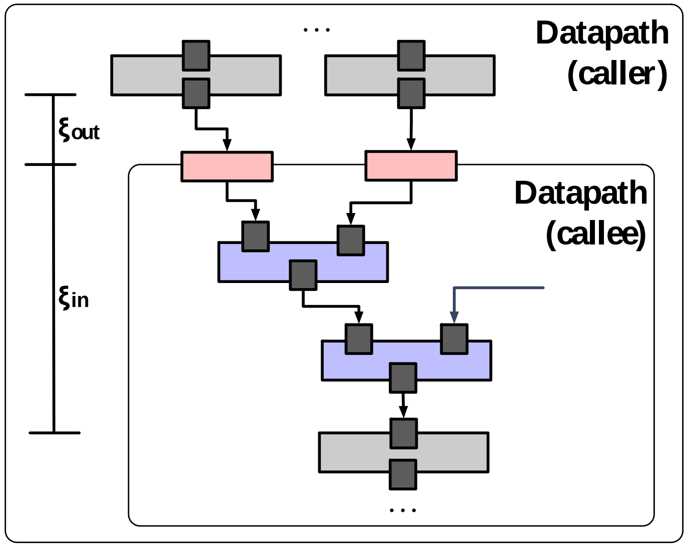
**Direct connection**



**Long external logic ($\xi_{out}$)**



**Long internal logic ($\xi_{in}$)**



## Dynamic Memory Allocation

- We require a **configurable memory space** to store the created data structures
  - *Memory management* is usually intrinsic within the algorithm

- We leverage **Memmgr**
  `https://github.com/eliben/code-for-blog/tree/master/2008/memmgr`
  - library for dynamic memory allocation to a configurable space with a synthesizable **`malloc`** function
  - *pre-allocated memory space* with a default size of 384 KB
  - memory space either in a local memory or in DRAM
  - memory component to manage *pointer-based requests*

## How to Connect the Memory Components?

- Two conflicting goals:
  - Maintain **flexibility** as in software (operations on pointers, dynamic resolution, etc.)
  - Increase **efficiency** as in hardware (exploit hardware parallelism whenever possible)

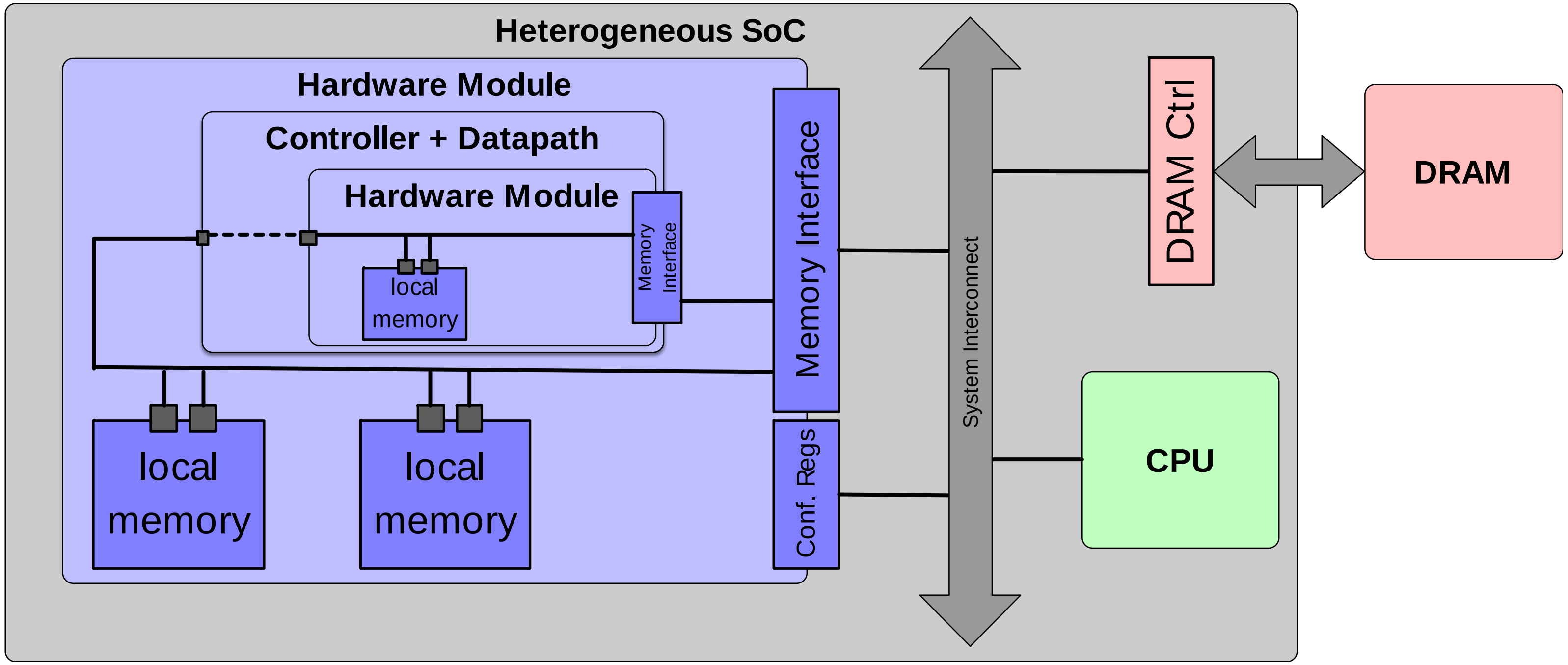
**Definition:**

***Points-to set***: set of data structures that can be potentially accessed by the pointer used in a memory operation

- This information is obtained during compilation by means of (sophisticated) **alias analysis**

# Flexible Memory Architecture

- ❑ **Direct connections** to datapath operators if the operation is completely defined
- ❑ **Internal memory bus** to connect all memory components potentially accessed by unresolved memory operations (*points-to set*)



## Memory Allocation

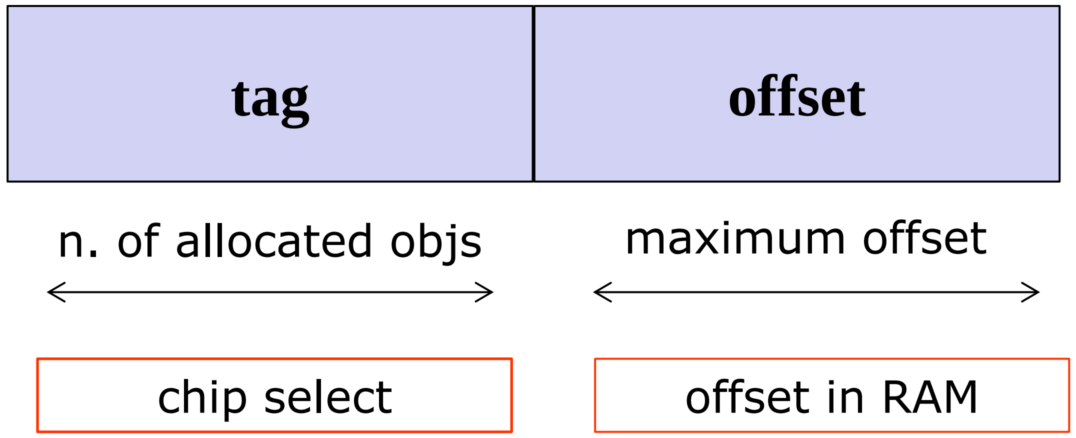
- **Memory analysis phase** to define an address for any
  - global scalar/aggregate variables,
  - local aggregate variables
  - local scalar variables used as argument of operator &

- Pointers are stored in standard registers
  - They can be considered as standard variables

- Load and store memory operations can be implemented as standard datapath operations
  - Connected to dedicated memory resources (memory controllers)

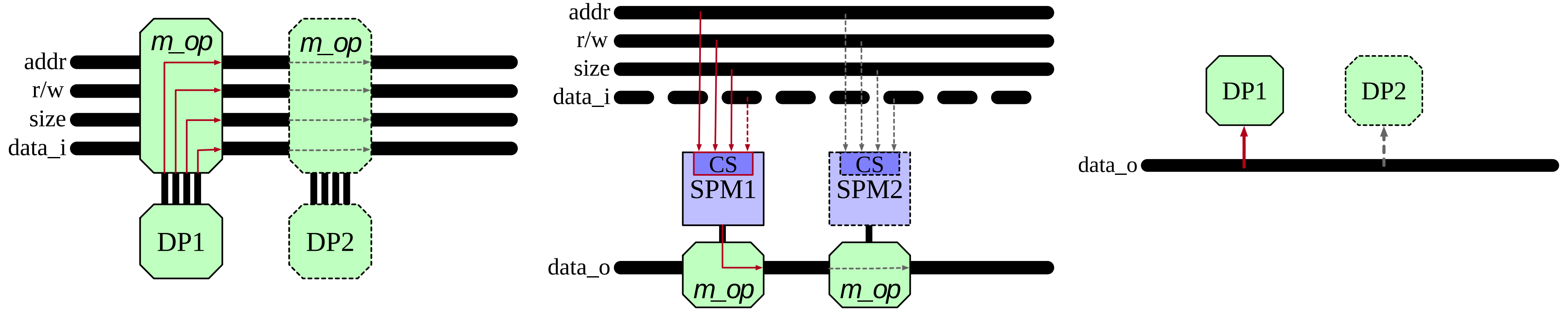
## Address Decoding

- We envision an **unique address space**, where memory data has an address associated with:
  - Contiguous or tag-based **address allocation**
  - Tag-based can simplify the control logic

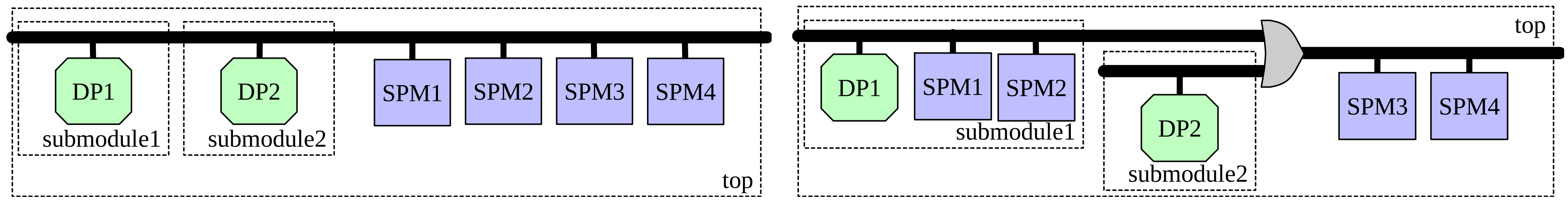


## Reducing the Logic Complexity

- Very powerful memory microarchitecture



- Possibility of creating long combinational paths with many memory components connected in chain
  - **Dominator-based analysis** and **bus merging** (OR gates) for localizing the memories as much as possible

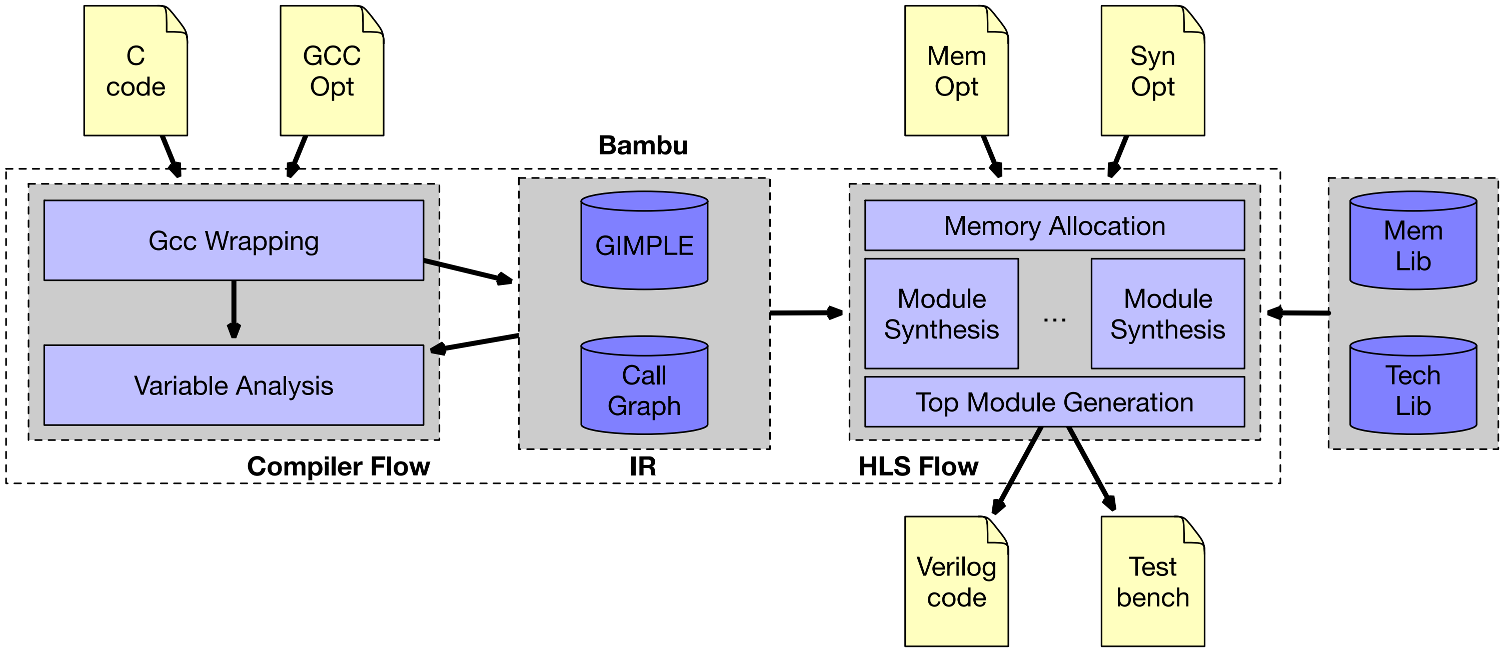


Daisy chain

Bus merging

# HLS-based Design Methodology

- **Automatic generation and optimization** fully implemented in Bambu
  - GCC for extracting **memory-related information**
  - Extensive set of **memory parameters** to explore many alternative configurations



## Memory-aware Optimizations

- **Reference/Naïve approach**
  - all data structures with standard memory components, instantiated in the top module and on the internal bus (**maximum flexibility**)

- **Privatization**
  - flow-sensitive pointer analysis to reduce the points-to set
  - resolved memory operations directly connected to the memories

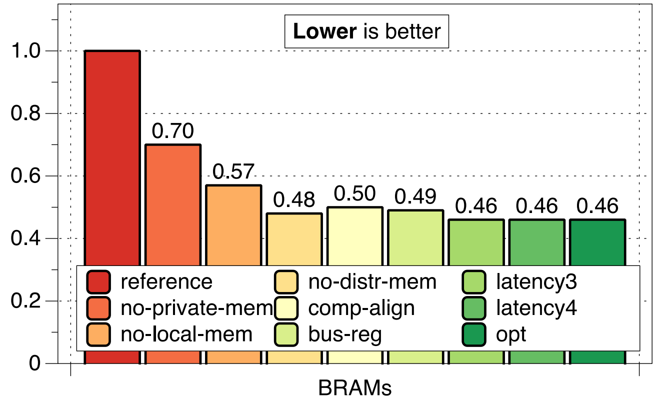
- Creation and duplication of **read-only memories**

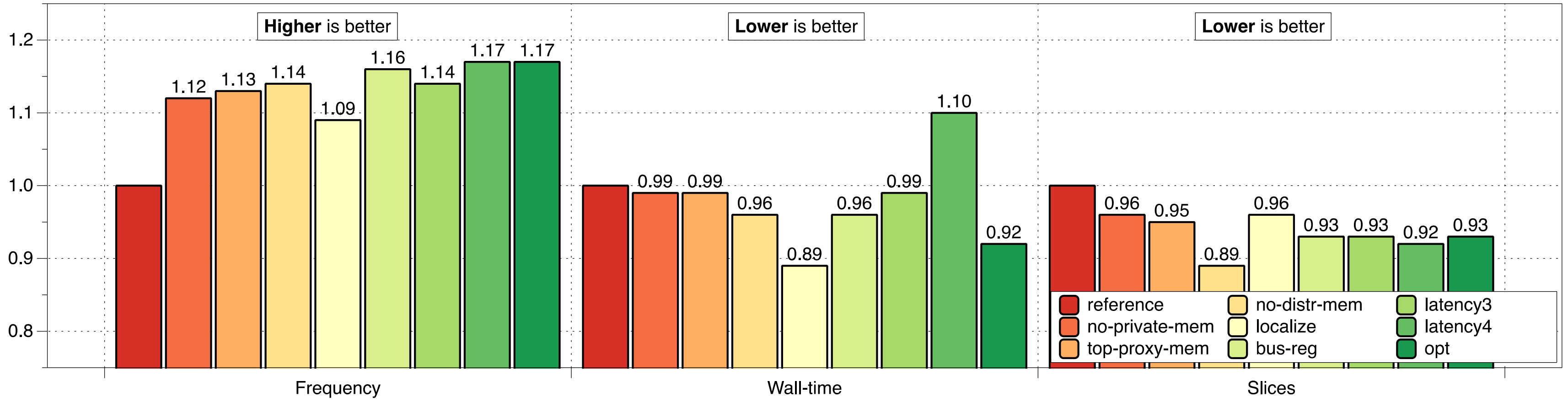
## Memory-aware Optimizations (cont'd)

- Conversion into **distributed memories**
  - configurable threshold
  - higher threshold for read-only memories (simpler)
- **Memory registering**
  - potential timing violations with complex memory operations and high target frequency
  - registers between datapath resources and memory ports
- **Localization**
  - dominator-based analysis to reduce long combinational paths

## Results on Xilinx

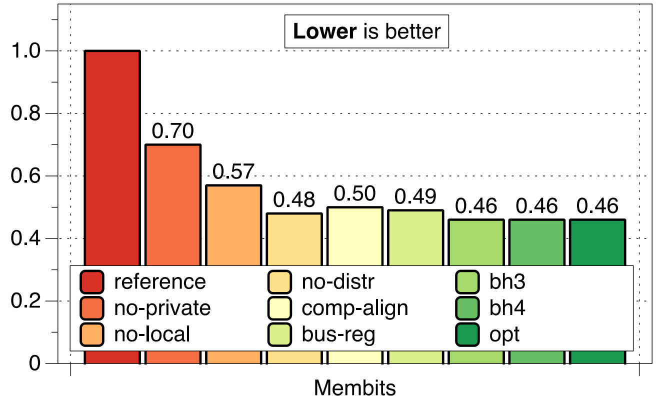
- Memory components can reach 312 MHz (XC7Z020) and 608 MHz (XC7VX690T)
- Selected applications from a recent survey on HLS tools (Nane et al. TCAD 2016)
  - Frequency of 400MHz
  - Up to 54% of memory reduction

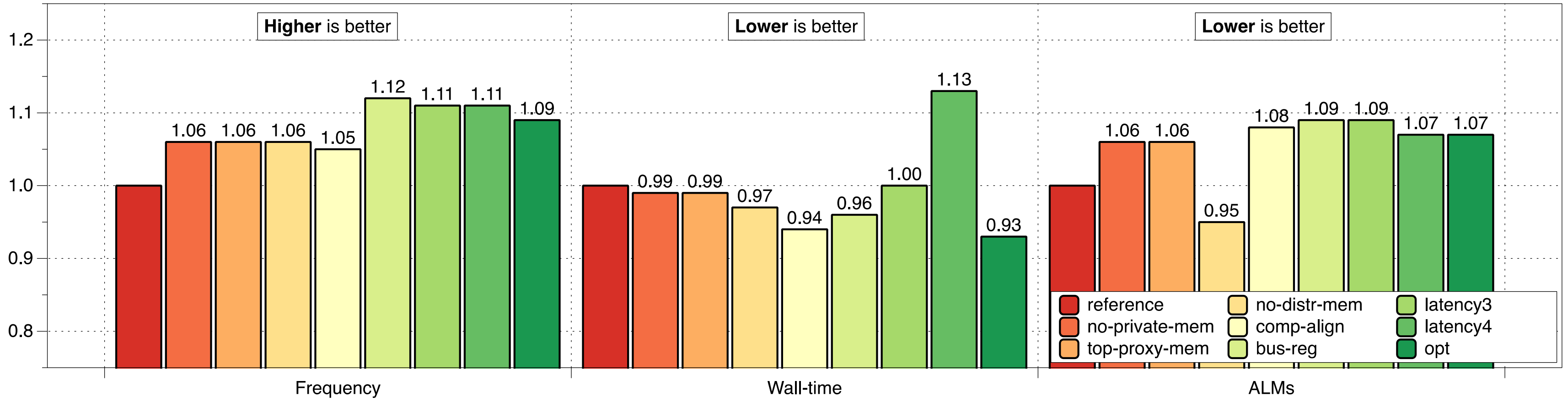




## Results on Intel-Altera

- Memory components can reach 600 MHz (Stratix-V)
- Selected applications from a recent survey on HLS tools (Nane et al. TCAD 2016)
  - Frequency of 400MHz
  - Up to 54% of memory reduction





## First example – memory allocation policy

### Explore different allocation policy

```
--memory-allocation-policy=<type>
        Set the policy for memory allocation. Possible values for the <type>
        argument are the following:
            ALL_BRAM           - all objects that need to be stored in memory
                                 are allocated on BRAMs (default)
            LSS                - all local variables, static variables and
                                 strings are allocated on BRAMs
            GSS                - all global variables, static variables and
                                 strings are allocated on BRAMs
            NO_BRAM            - all objects that need to be stored in memory
                                 are allocated on an external memory
            EXT_PIPELINED_BRAM - all objects that need to be stored in memory
                                 are allocated on an external pipelined memory
```

## Solution Hint

```
$ bambu adpcm.c --memory-allocation-policy=LSS
--clock-period=15 --simulate -v3
```

Look for the log section:

```
Memory allocation information:
    Variable external to the top module: test_result - 25438 -
test_result
      Id: 25438
      Base Address: 1073741824
      Size: 400
      Is a Read Only Memory
      Used &(object)
      Number of functions in which is used: 1
      Maximum number of references per function: 1
      Maximum number of loads per function: 1
    …
```

## Solution

```
$ bambu adpcm.c --memory-allocation-policy=ALL_BRAM
--clock-period=15 --simulate -v3

$ bambu adpcm.c --memory-allocation-policy=LSS
--clock-period=15 --simulate -v3

$ bambu adpcm.c --memory-allocation-policy=GSS
--clock-period=15 --simulate -v3

$ bambu adpcm.c --memory-allocation-policy=NO_BRAM
--clock-period=15 --simulate -v3

$ bambu adpcm.c
--memory-allocation-policy=EXT_PIPELINED_BRAM
--clock-period=15 --simulate -v3
```

# Second example – Multi-channel design space exploration

```
--channels-type=<type>
       Set the type of memory connections.
       Possible values for <type> are:
           MEM_ACC_11 - the accesses to the memory have a single direct
                        connection or a single indirect connection (default)
           MEM_ACC_N1 - the accesses to the memory have n parallel direct
                        connections or a single indirect connection
           MEM_ACC_NN - the accesses to the memory have n parallel direct
                        connections or n parallel indirect connections

  --channels-number=<n>
       Define the number of parallel direct or indirect accesses.
```

- When BRAMs are involved only two ports at maximum could be given
- When option
  `--memory-allocation-policy=EXT_PIPELINED_BRAM`
  is given the number of channels could be greater than 2

## Solution Hint

```
$ bambu adpcm.c --channels-type=MEM_ACC_NN --memory-
allocation-policy=EXT_PIPELINED_BRAM --channels-number=4
--clock-period=15 --simulate –v3
```

Look how long it take the simulation.

Consider `-fwhole-program` option

## Third example – Control the Load/Store latency

```
--memory-ctrl-type=type
       Define which type of memory controller is used.
       Possible values for the <type> argument are the following:
           D00 - no extra delay (default)
           D10 - 1 clock cycle extra-delay for LOAD, 0 for STORE
           D11 - 1 clock cycle extra-delay for LOAD, 1 for STORE
           D21 - 2 clock cycle extra-delay for LOAD, 1 for STORE

--bram-high-latency=[3,4]
       Assume a 'high latency bram'-'faster clock frequency'
       block RAM memory based architectures:
       3 => LOAD(II=1,L=3) STORE(1).
       4 => LOAD(II=1,L=4) STORE(II=1,L=2).

--mem-delay-read=value
       Define the external memory latency when LOAD are performed (default 2).

--mem-delay-write=value
       Define the external memory latency when LOAD are performed (default 1).
```

## Solution Hint

```
$ bambu mips.c --memory-ctrl-type=D21 --channels-
type=MEM_ACC_NN --memory-allocation-policy=EXT_PIPELINED_BRAM
--channels-number=4
--clock-period=15 --simulate -v3
```

Look how long it take the simulation.

```
$ bambu mips.c --bram-high-latency=4 --channels-
type=MEM_ACC_NN --clock-period=15 --simulate -v3
```

Look how long it take the simulation.

## Fourth example – Control asynchronous memories inference

```
--do-not-use-asynchronous-memories
       Do not add asynchronous memories to the possible set
       of memories used by bambu during the memory allocation
       step.

--distram-threshold=value
       Define the threshold in bitsize used to infer
       DISTRIBUTED/ASYNCHRONOUS RAMs (default 256).
```

## Solution Hint

```
$ bambu mips.c --do-not-use-asynchronous-memories -fwhole-
program --clock-period=15 --simulate -v3
```

Look how long it take the simulation.

```
$ bambu mips.c --distram-threshold=1024 -fwhole-program --
clock-period=15 --simulate -v3
```

Look how long it take the simulation.

# Fifth – Alignment hints

```
--unaligned-access
       Use only memories supporting unaligned accesses.

--aligned-access
       Assume that all accesses are aligned and so only
       memories supporting aligned accesses are used.
```

## Solution Hint

```
$ bambu mips.c --unaligned-access -fwhole-program --clock-
period=15 --simulate -v3
```

Look how long it take the simulation.

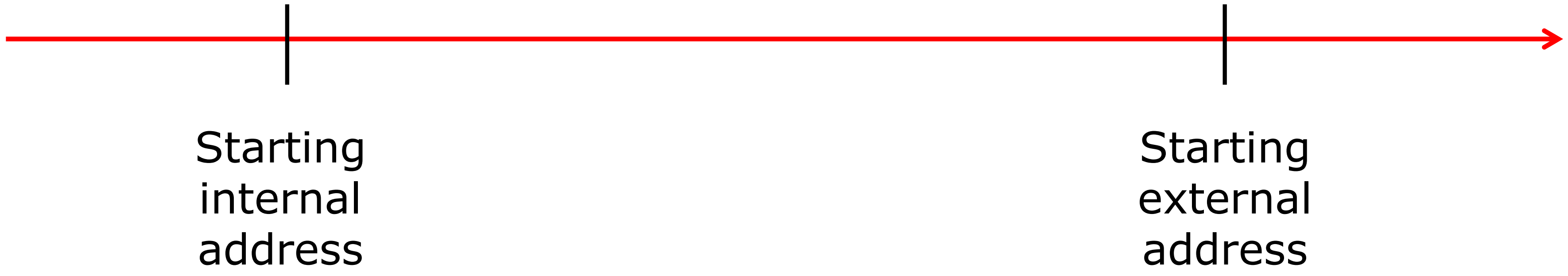
```
$ bambu mips.c --aligned-access -fwhole-program --clock-
period=15 --simulate -v3
```

Look how long it take the simulation.

## Sixth example – customize memory layout

```
--base-address=address
     Define the starting address for objects allocated externally to the top
     module.

--initial-internal-address=address
     Define the starting address for the objects allocated internally to the
     top module.
```



## Solution Hint

```
$ bambu mips.c --base-address=1024 --memory-allocation-policy=LSS --clock-period=15 --simulate -v3
```

Look how long it take the simulation.

```
$ bambu mips.c --initial-internal-address=0 --base-address=1024 --memory-allocation-policy=LSS --clock-period=15 --simulate -v3
```

Look how long it take the simulation.

# Other options

```
--sparse-memory[=on/off]
        Control how the memory allocation happens.
            on - allocate the data in addresses which reduce the decoding logic (default)
           off - allocate the data in a contiguous addresses.

--serialize-memory-accesses
        Serialize the memory accesses using the GCC virtual use-def chains
        without taking into account any alias analysis information.

--do-not-chain-memories
        When enabled LOADs and STOREs will not be chained with other
        operations.

--rom-duplication
        Assume that read-only memories can be duplicated in case timing requires.


--do-not-expose-globals
        All global variables are considered local to the compilation units.

--data-bus-bitsize=<bitsize>
        Set the bitsize of the external data bus.

--addr-bus-bitsize=<bitsize>
        Set the bitsize of the external address bus.
```