POLITECNICO DI MILANO

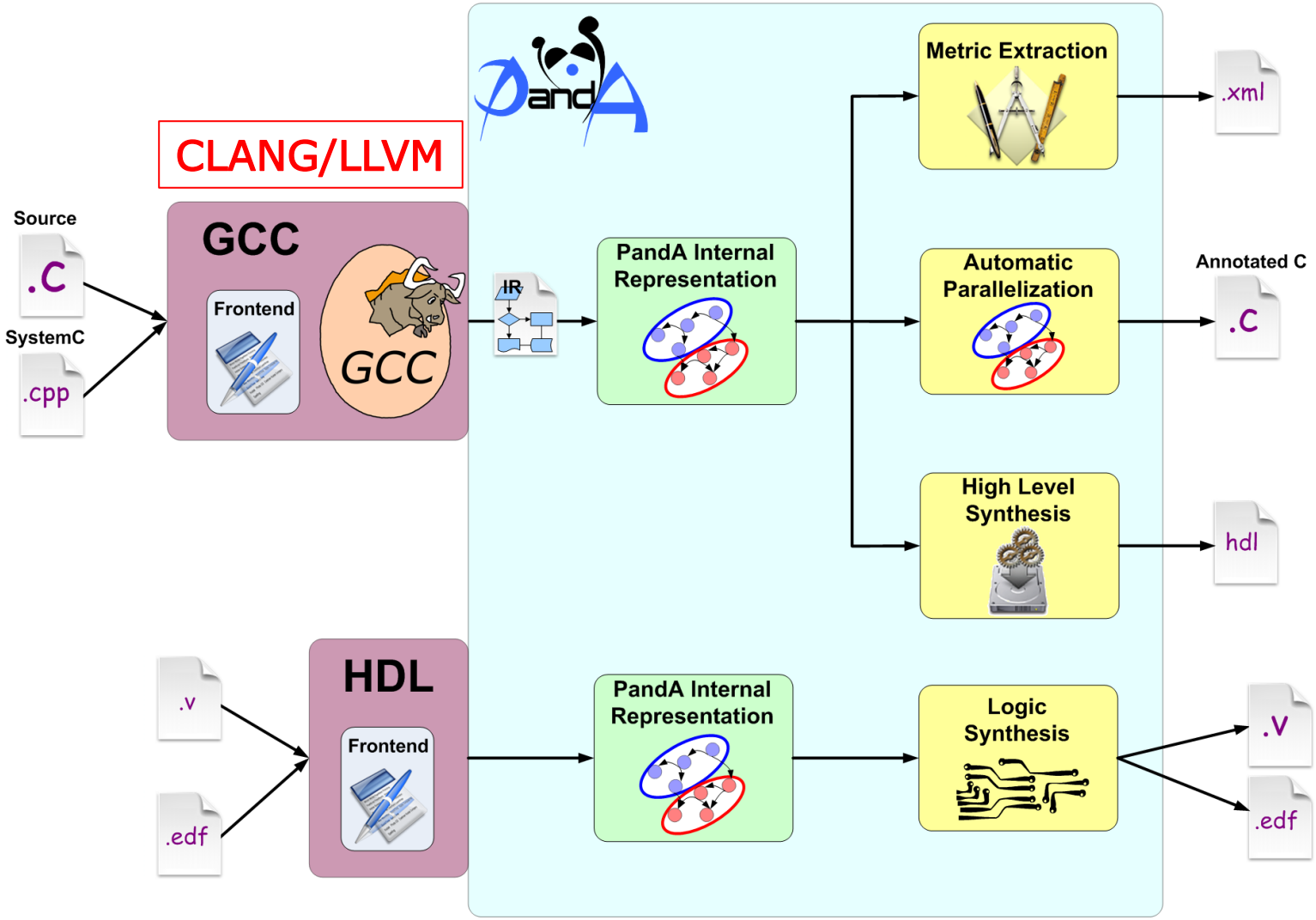
Bambu: FPGA Programming for Complex Parallel Applications

Fabrizio Ferrandi <fabrizio.ferrandi@polimi.it >
Politecnico di Milano

A bit of history

- PandA framework development started on 2004 as a support research infrastructure for PoliMi in the context of ICODES – FP6-IST EU-funded project
  - Parsing and analysis of TLM 2.0 SystemC descriptions (gcc v.3.5)
- In the hArtes EU-funded project (2006-2010), it was used to
  - Analyzing generic C-based application annotated with pragmas (OpenMP)
  - Extracting parallel tasks
  - Estimating performance of embedded app
  - C-to-C rewriting
- Later, in Synaptic (2009-2013) and in Faster (2011-2014) EU-funded projects, logic- and high-level synthesis has been extended
  - Bambu (HLS tool) was first released in March 2012.
- ESA funded many research on code predictability analysis, performance analysis, and integration of HLS in model-based design flows.

Framework Overview



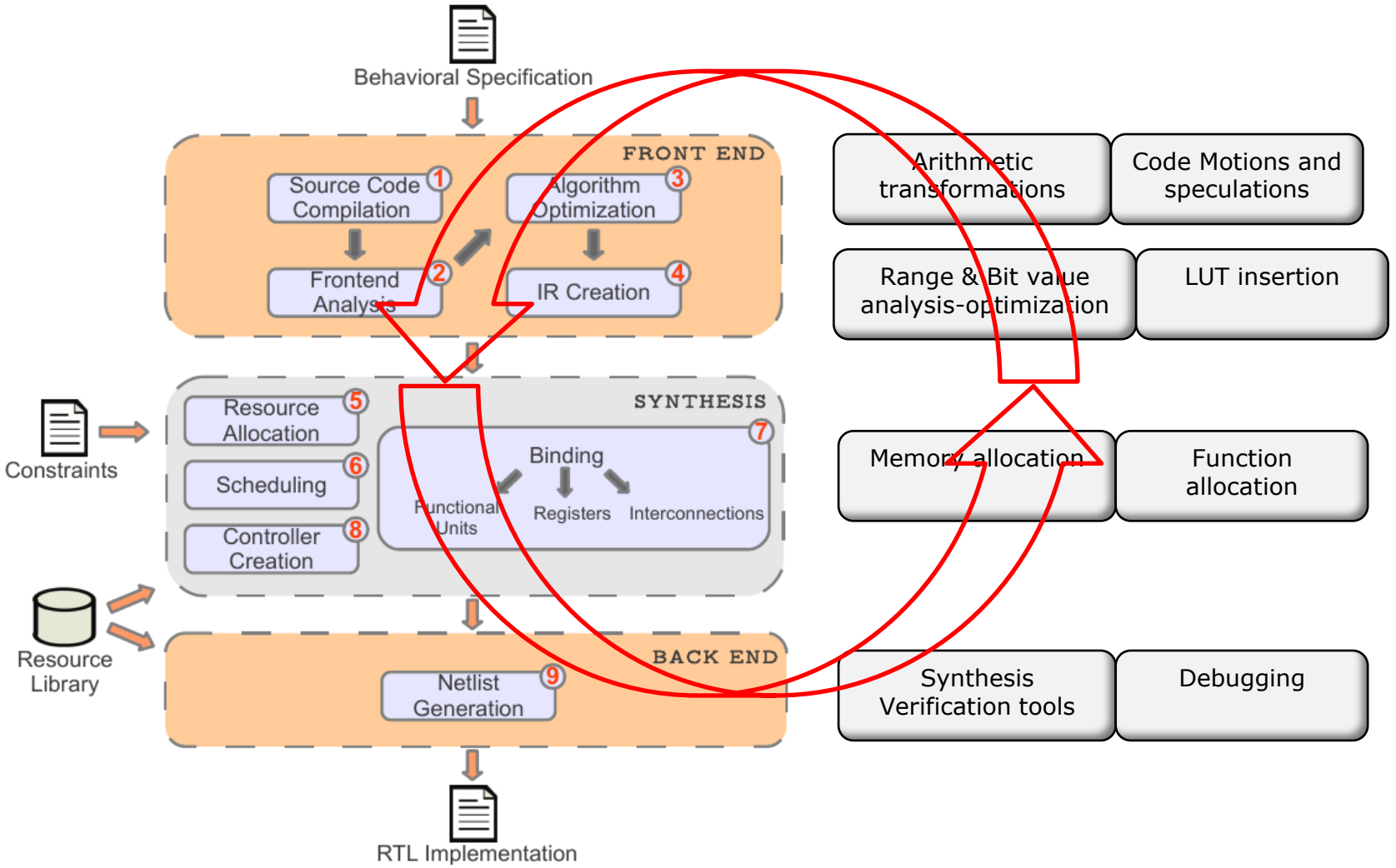
Bambu: an example of modern HLS tools

- HLS tool developed at Politecnico di Milano (Italy) within the PandA framework
  - Available under GPL v3 at
    - http://panda.dei.polimi.it/
    - https://github.com/ferrandi/PandA-bambu
- Example features
  - Front-end Input: interfacing with GCC/CLANG-LLVM for parsing C code
    - Complete support for ANSI C (except for recursion)
      - Support for pointers, user-defined data types, built-in C functions, etc..
    - Source code optimizations
      - may alias analysis, dead-code elimination, hoisting, loop optimizations, etc...
  - Target-aware synthesis
    - Characterization of the technology library based on target device
  - Verification
    - Integrated testbench generation and simulation
      - automated interaction with Iverilog, Verilator, Xilinx Isim, Xilinx Xsim, Mentor Modelsim
  - Back-end: Automated interaction with commercial synthesis tools
    - FPGA: Xilinx ISE, Xilinx Vivado, Altera Quartus, Lattice Diamond
    - ASIC: Synopsis Design Compiler

Synthesis per function

- one component per function
  - function interface
  - start and done
  - parameter passing
    - wires
    - memory interaction
- hierarchy based on call graph
  - no-recursion
  - proxy
- option
  - `-fwhole-program`

Bambu Overview



Modular Framework based on the specialization of **HLS_step**

Bambu: command-line interface

- Minimal command
  - `$ bambu filename.c`
- Controlling the clock period (100Mhz)
  - `$ bambu filename.c --clock-period=10`
- Select the device
  - `$ bambu filename.c –device-name=xc7z020,-1,clg484,VVD`

Subset of synthesizable C (1)

- We support what standard compilers accept as input (CLANG/LLVM and GCC)
- Supported features:
  - Expressions of any kind: arithmetic, logical, bitwise, relational, conditional, comma-based expressions.
  - Types: integers, single- and double-precision floating point, _Bool and Complex, struct-or-union, bitfields, enum, typedef, pointers and arrays, type qualifiers.
  - Variable declarations, initialization, storage-specifiers
  - Functions definition and declaration, extern or static, pointer to functions, parameters passed by copy or reference, tail recursive functions.
  - Statements and blocks: labeled (case), compound, expression, selection (if,switch), iteration(while,do,for), jump (goto,continue,break,return)
  - All preprocessor directives
  - Unaligned memory accesses and dynamic pointers resolution
  - GCC vectorization

Subset not supported

- struct returned by copy
- Non-tailing recursive functions

libbambu

- assert, puts, putchar, read, open, close, write, printf, exit, abort

- libc functions: bswap32,memcmp, memcpy, memmove, memset, malloc, free, memalign, alloca_with_align, calloc, bcopy, bzero, memchr, mempcpy, memrchr, rawmemchr, stpcpy, stpncpy, strcasecmp, strcasestr, strcat, strchr, strchrnul, strcmp, strcpy, strcspn, strdup, strlen, strncasecmp, strncat, strncmp, strncpy, strndup, strnlen, strpbrk, strrchr, strsep, strspn, strstr, strtok

- libm functions: acos, acosh, asin, asinh, atan, atan2, atanh, cbrt, ceil, cexpi, copysign, cos, cosh, drem, erf, exp, exp10, expm1, fabs, fdim, finite, floor, lfloor, fma, fmax, fmin, fmod, fpclassify, frexp, gamma, lgamma, tgamma, hypot, ilogb, infinity, isinf, isnan, j0, j1, jn, ldexp, log, log2, log10, log1p, modf, nan, nearbyint, nextafter, pow, pow10, remainder, remquo, rint, lrint, llrint, round, lround, llround, scalb, scalbln, scalbn, signbit, significand, sin, sincos, sinh, sqrt, tan, tanh, trunc.

synthesis of multiple files

- `bambu` uses the text-based IR and the builtin linker to build a single in-memory representation and perform HLS
  - `$ bambu file1.c file2.c --top-fname=fname`
- In case the option `--top-fname` is not used, `bambu` automatically identifies which function can act as the top one.

Synthesis per function: interface

- Based on C description
  - Global variables
  - Top function parameters
- No restriction on top function signature
- C semantic
  - Scalar parameters: input
  - struct/union parameters: input
  - array/pointer: input/output
  - return value: output
  - Global variables: input/output

Interface analysis

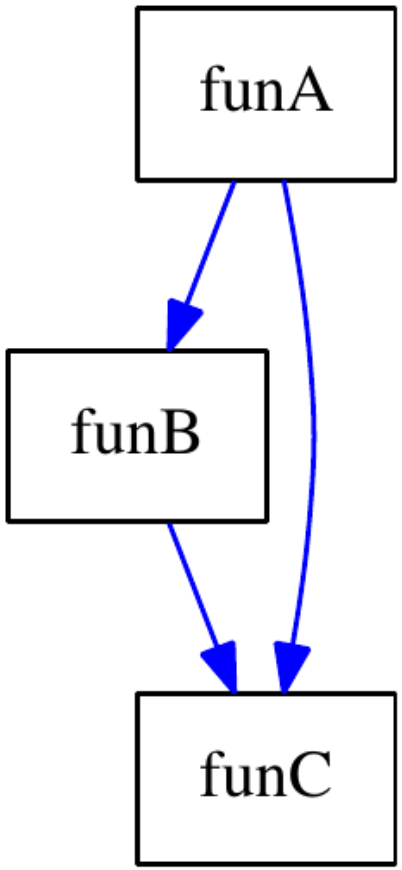

```
int global;

float my_top_function(int * first, struct s second, int
third[10], float * fourth)
{…}
```

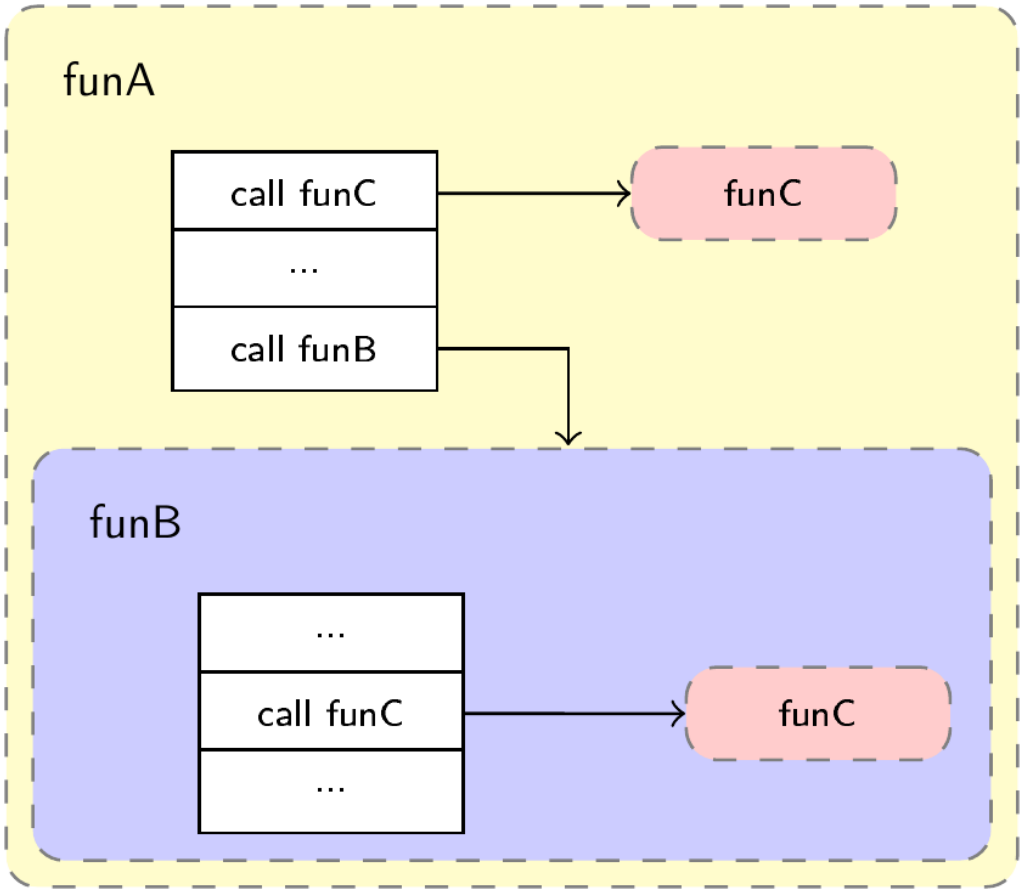
- Input parameters:
  - Second
- Input/ouptut parameters:
  - First, third, fourth
- Ouptut parameters:
  - Return value
- Shared memory:
  - global

Synthesis per function: hierarchy

- One component per function

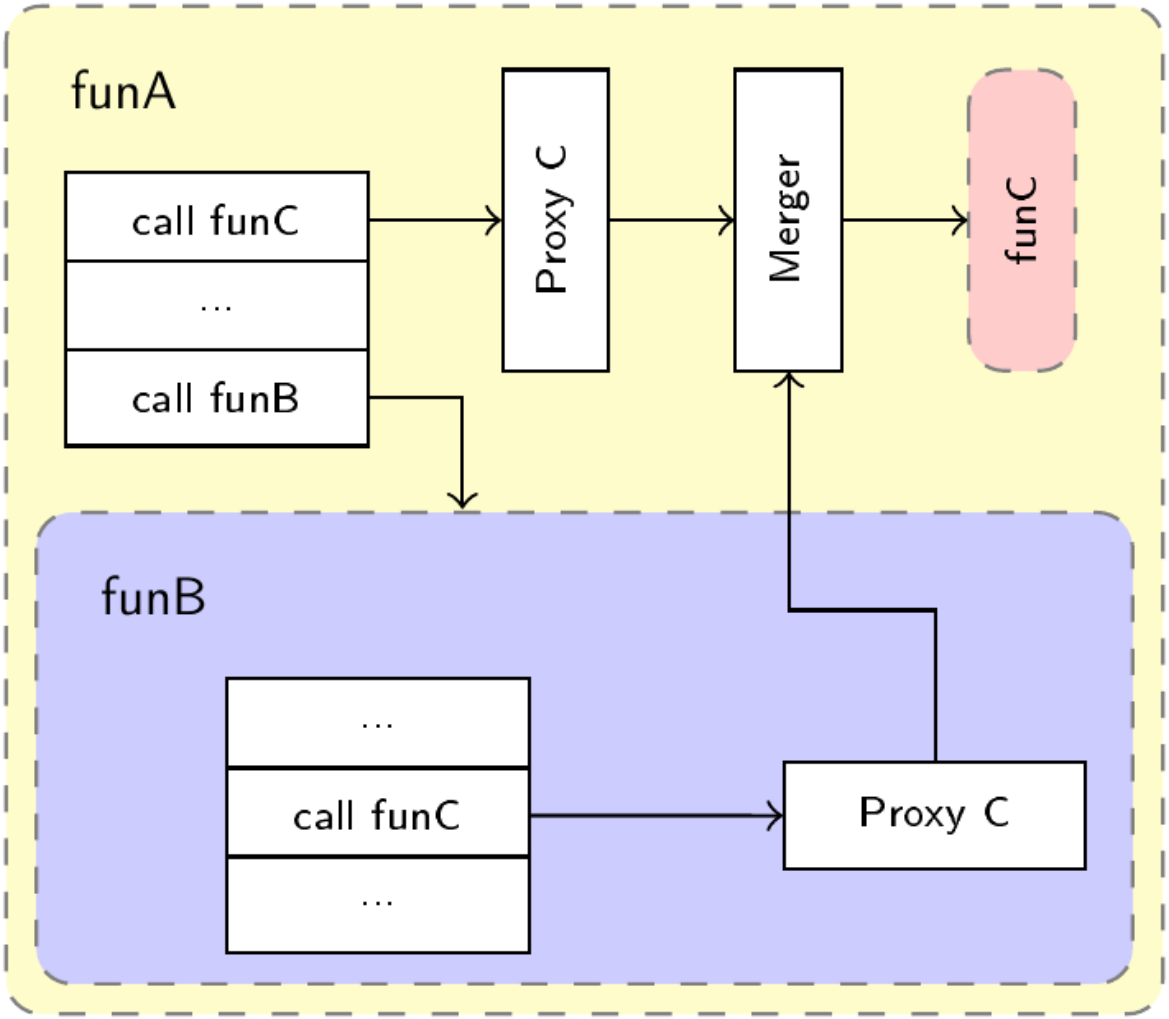


call graph



RTL hierarchy

Synthesis per function: proxy



Option: `--do-not-expose-globals`

- bambu assumes that all global variables could be accessed by external CPU/accelerators
- It is possible to change this default with option:
  - `--do-not-expose-globals`
  - All global variables are considered local to the compilation units as it they are declared `static`.

Integration of hand- written components in the HLS flow

- Function mapped on IPs has to be declared as extern:
  - `extern void module1(uint32_t input1, uint16_t input2, module1_output_t *outputs);`
- C code has to be passed with the following option
  - `--C-no-parse=module1.c,…`
- Binding between function **module1** and component **module1** has to be specified with a XML file and passed as an option to bambu
  - `$ bambu … module_lib.xml`
- Check these examples:
  - `examples/IP_integration`
  - `examples/breakout`
  - `examples/pong`
  - `examples/led_example`

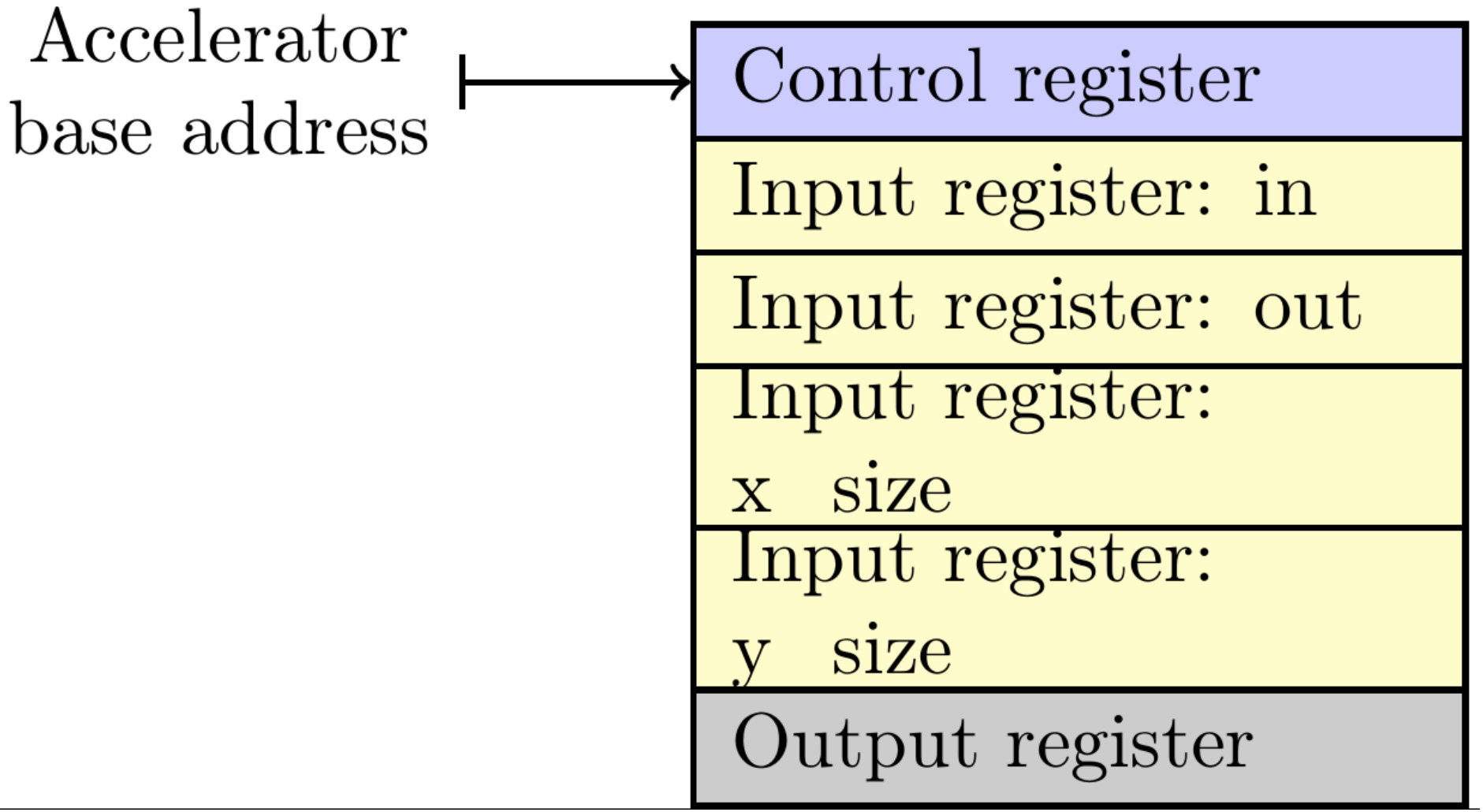
Synthesis of function pointers


```
int laplacian(char *, char *, int, int);
int make_inverse_image(char *, char *, int, int);
int sharpen(char *, char *, int, int);
int sobel(char *, char *, int, int);

int (*pipeline[MAX_DEPTH])(char *, char *, int, int);

void UserApp(char *in, char *out, int x_size, int y_size) {
  // ...
  // Pipeline configuration using function pointers
  add_filter(0, make_inverse_image);
  add_filter(1, sharpen);
  // ...
  // execute is synthesized in hardware
  execute(in, out, x_size, y_size);
}
void execute(char *in, char *out, int x_size, int y_size) {
  int i = 0;
  for (i = 0; i < MAX_PIPELINE_DEPTH; i++) {
    if (pipeline[i] == 0) break;
    // here other hw accelerator are called
    // using function pointers
    int res = pipeline[i](in, out, x_size, y_size);
    if (res != 0) return;
    swap(in, out);
  }
  move_if_odd(i, in, out);
}
```

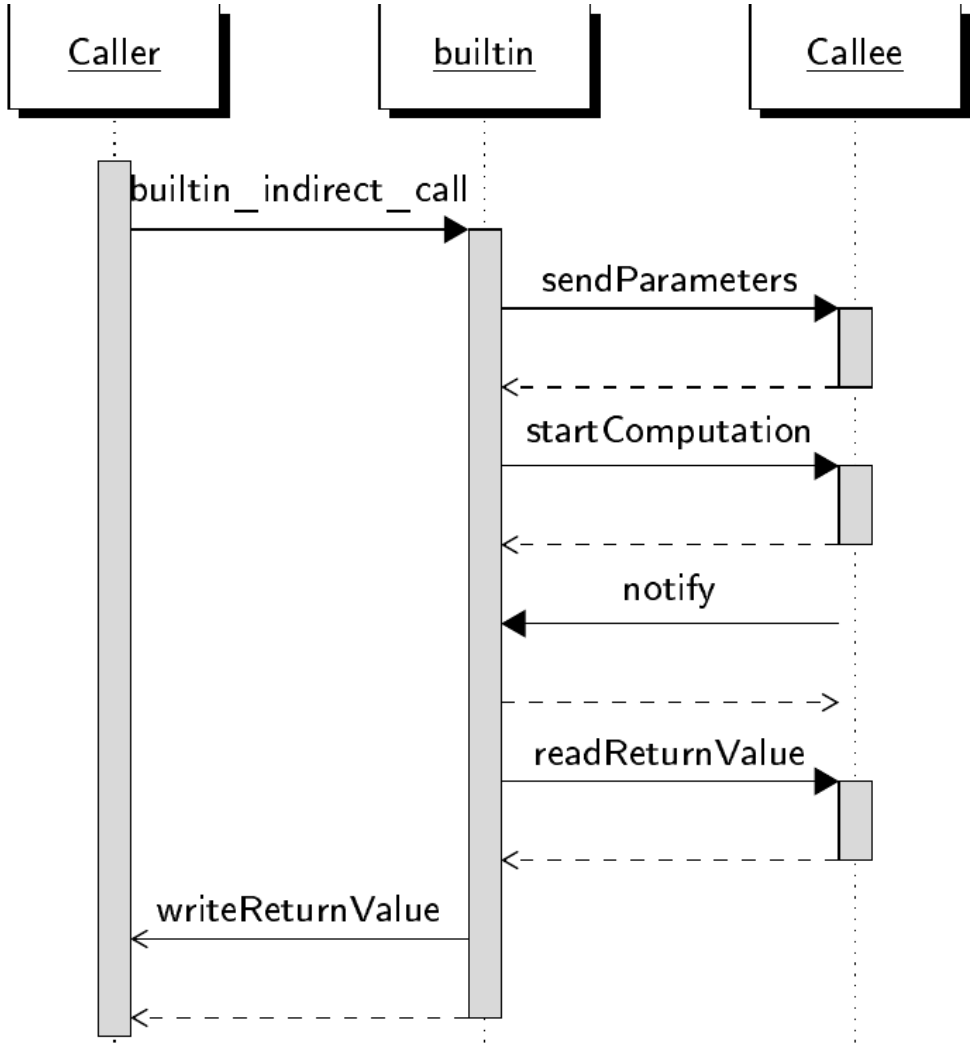
Adding a memory mapped interface to filters



Code transformation performed


```
void execute(char *in, char *out, int x_size, int y_size) {
  int i = 0;
  for (i = 0; i < MAX_PIPELINE_DEPTH; i++) {
    if (pipeline[i] == 0) break;
    // here other hw accelerator are called
    // using function pointers
    __builtin_indirect_call(
        pipeline[i], 1, in, out, x_size, y_size, &res);
    if (res != 0) return;
    swap(in, out);
  }
  move_if_odd(i, in, out);
}
```

Sequence diagram for function indirect call



Call mechanism complexity: #cycles=Wl(Np+1)+lhs(Wl+Rl)

C++ support

- Support of c++ is ongoing:
  - templates
  - C++11 and beyond
  - ac_types from Mentor Graphics could be used

```
#include <algorithm>
int gcd(int x, int y )
 {
        if( x < y )
           std::swap( x, y );

        while( y > 0 )
        {
           int f = x % y;
           x = y;
           y = f;
        }
        return x;
 }
```

OpenMP support

- #pragma omp simd is supported

```
#pragma omp declare simd
void add (int accelnum, int startidx, int endidx)
{
   int sum = 0;
   int i;
   for (i = 0; i < OPS_PER_ACCEL; i++) {
      sum += array[i+startidx];
   }
   output[accelnum] = sum;
}

main() {
...
#pragma omp simd
   for (i = 0; i < NUM_ACCELS; i++)
   {
      add(i, i * OPS_PER_ACCEL, (i + 1)*OPS_PER_ACCEL);
   }
```

- OmpMP support in progress…

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

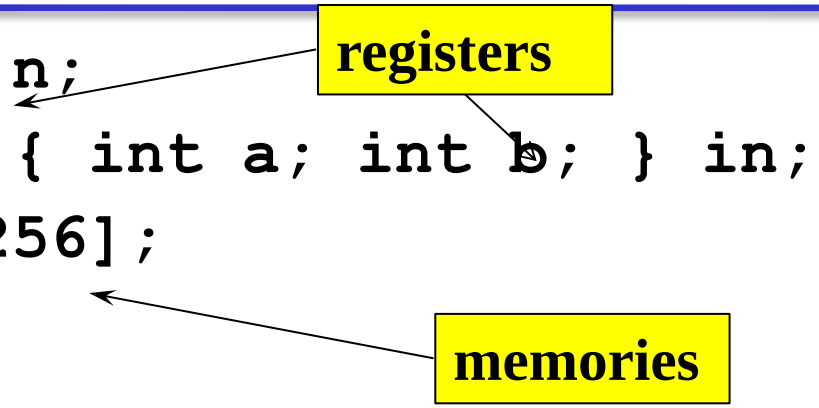
How to efficiently implement all memory operations?

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




```
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];
…
```

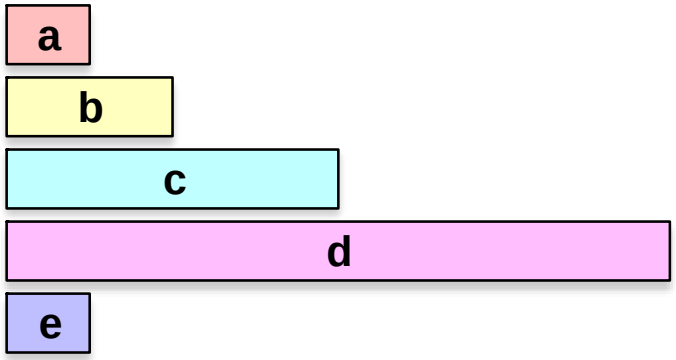
Software Semantics

- **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
```



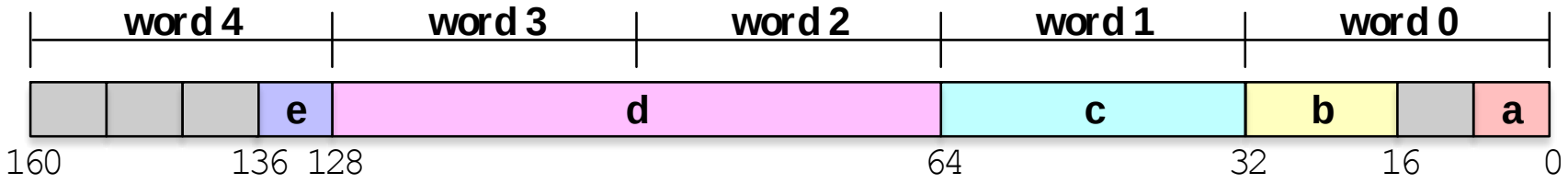
- How to *efficiently* store an array of **`mystruct`** structures?
- How to *efficiently* implement memory operations?

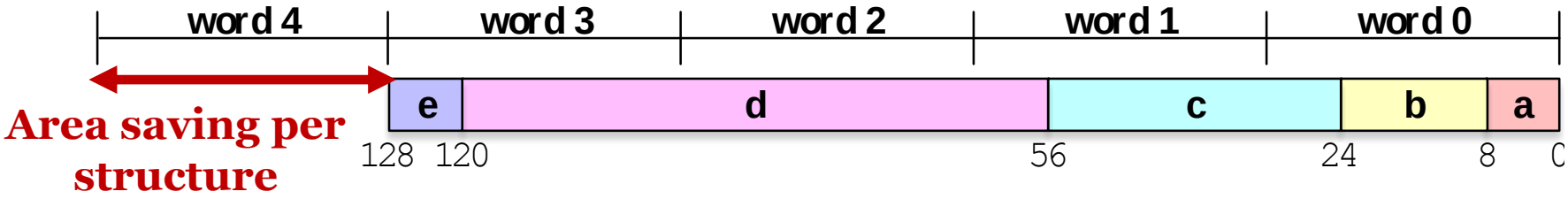
Reduce the memory footprint without compromising the execution latency

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

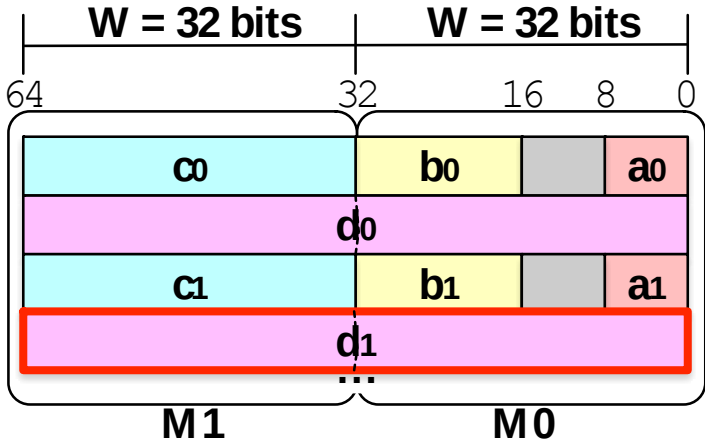
- Alternative solutions:
  - Vivado HLS decomposes the *array of structures* into *multiple arrays of single fields*
  - LegUp uses *subarrays* as large as the maximum field (support only for aligned accesses)

- 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

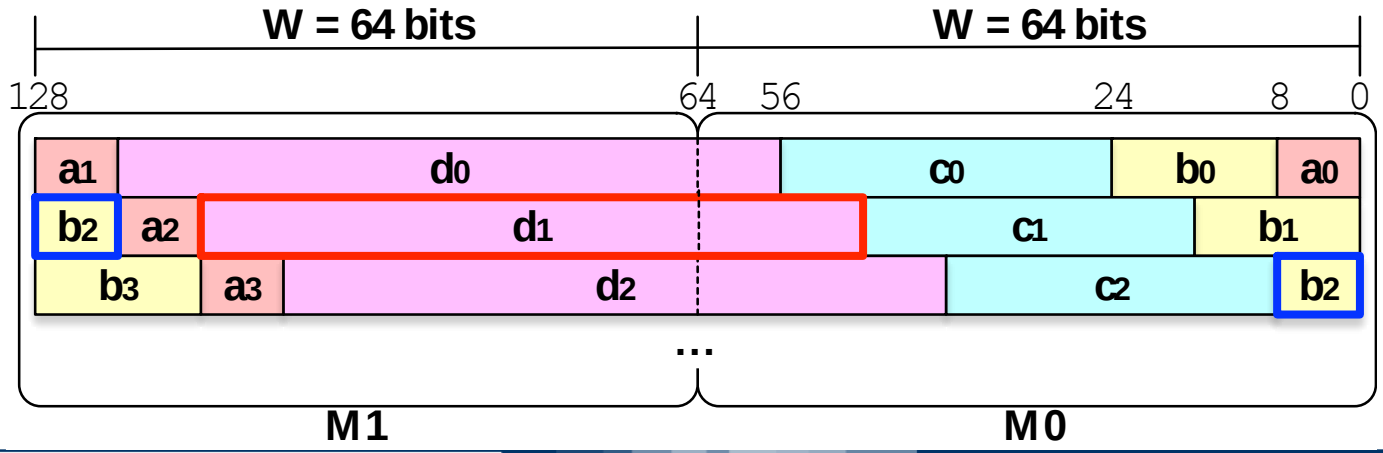
Memory Layouts

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

Aligned structures
(W = SMAX/2)



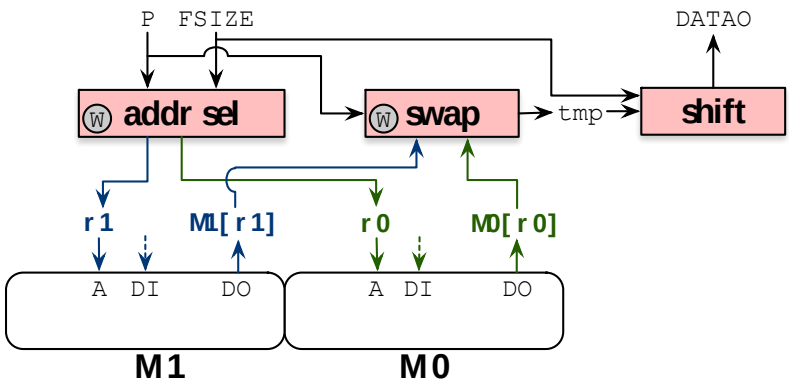
Unaligned structures
(W = SMAX)



Read and Write Operations

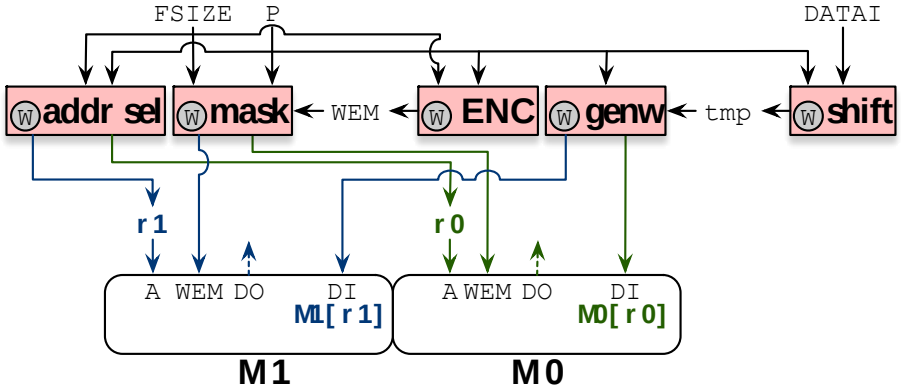
- **Unique microarchitecture** for both aligned and unaligned accesses

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]}
```

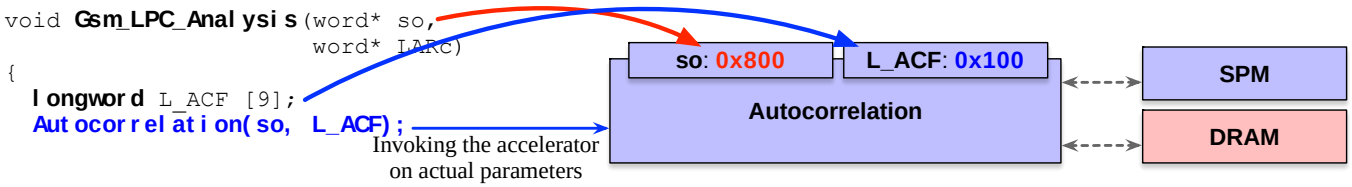
Write operations




```
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



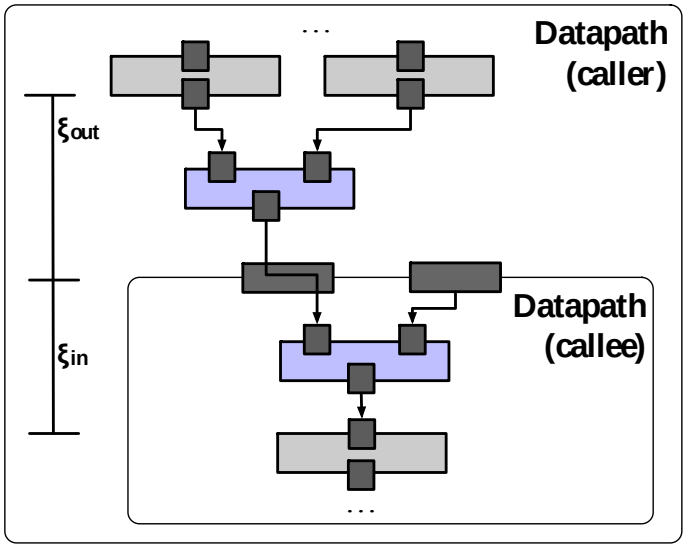
When to insert registers to avoid long critical paths?

- We introduce input registers when

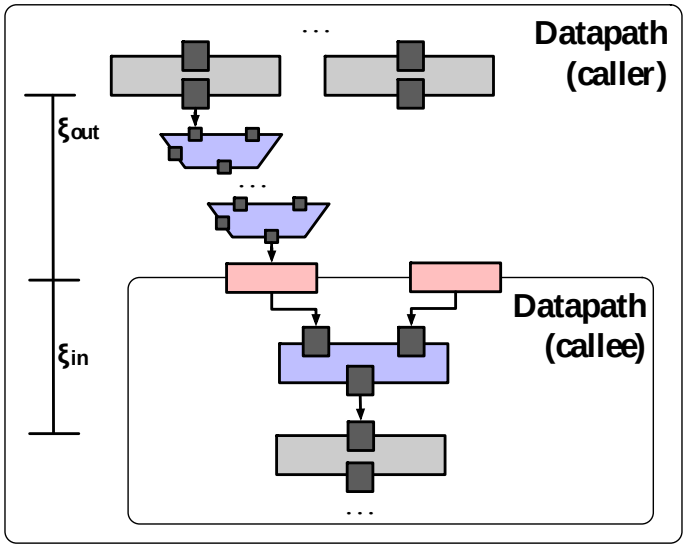
$$(\xi_{out} + \xi_{in}) > T * \beta$$

Some Examples of Input Registering

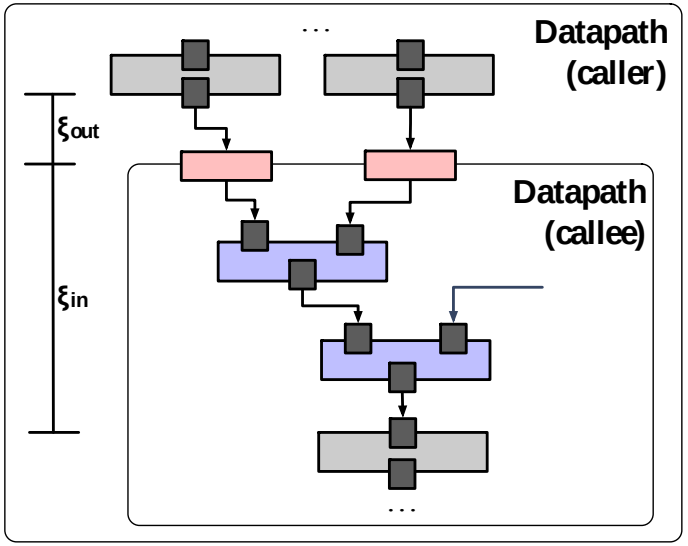
Direct connection



Long external logic (ξ_{out})



Long internal logic (ξ_{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)

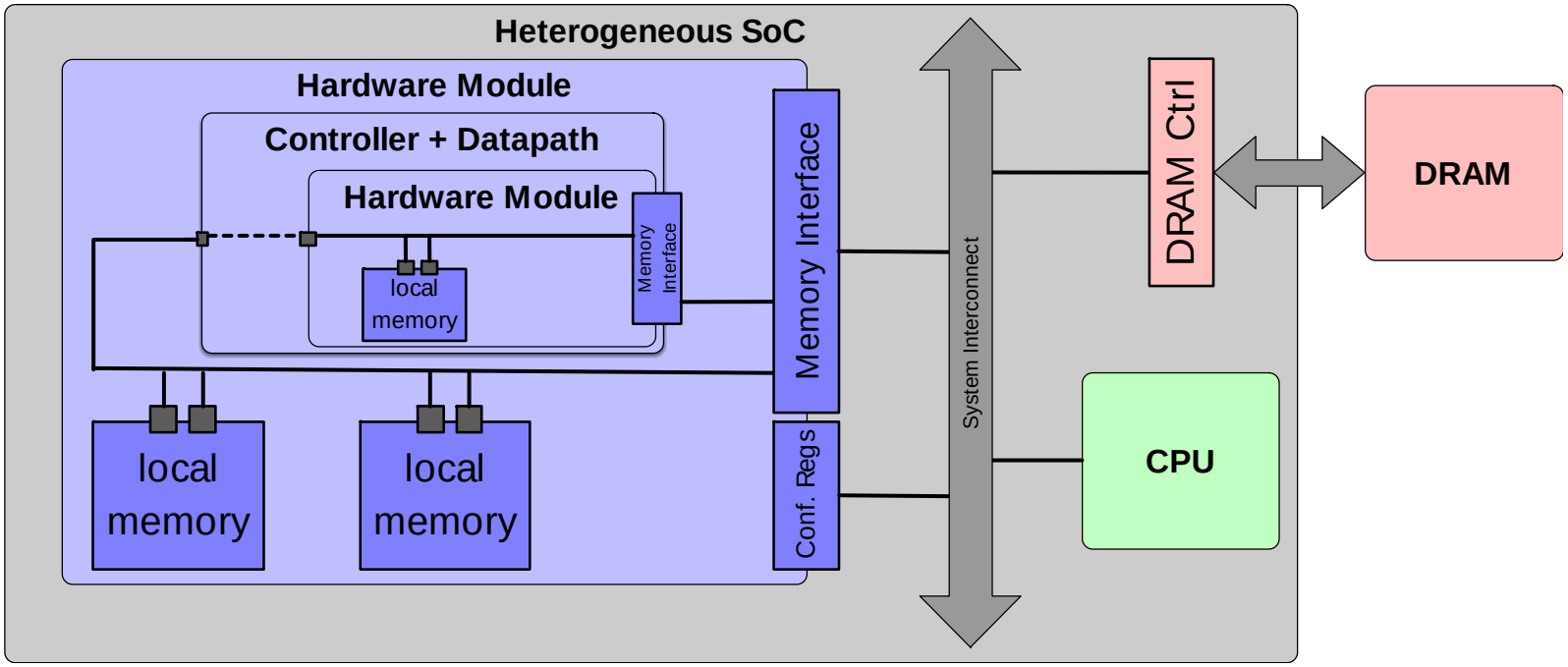
Definition:

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*)

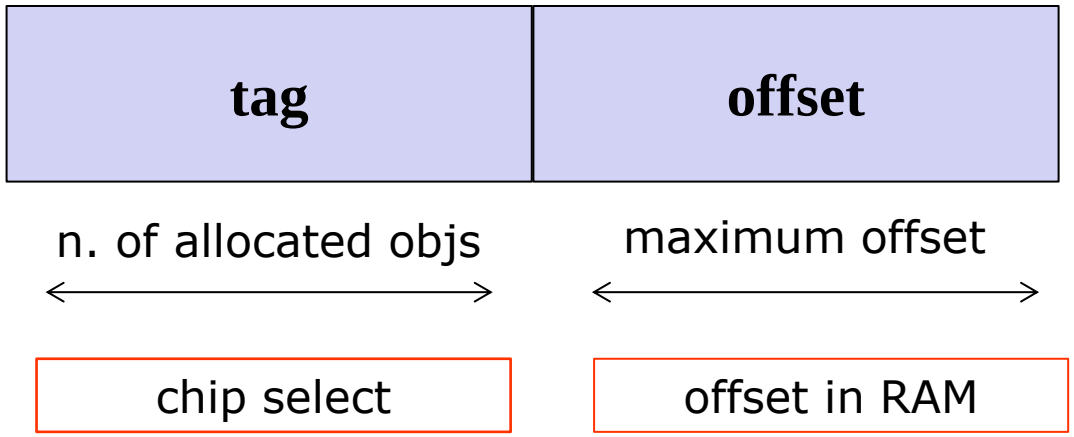


Memory Allocation

- **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)

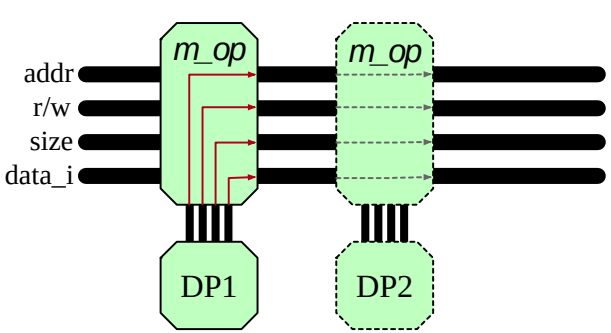
Address Decoding

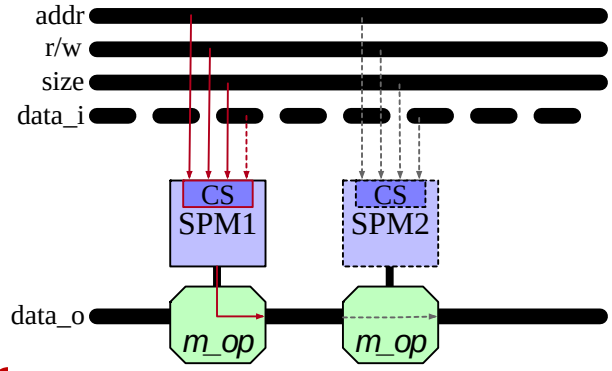
- We envision an **unique address space**, where memory data has an address associated with:
  - Contiguous or tag-based **address allocation**
  - Tag-based can simplify the control logic

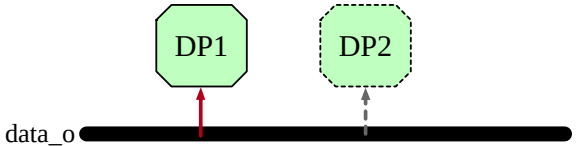


Reducing the Logic Complexity

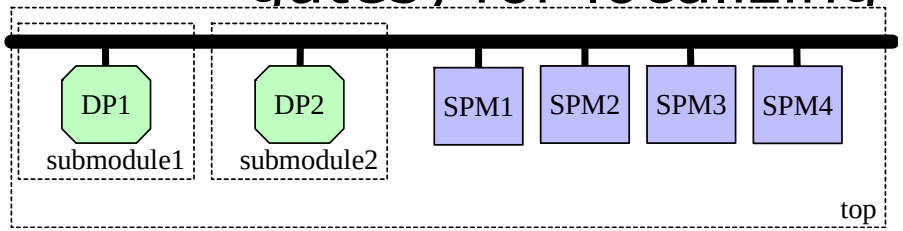
- Very powerful memory microarchitecture



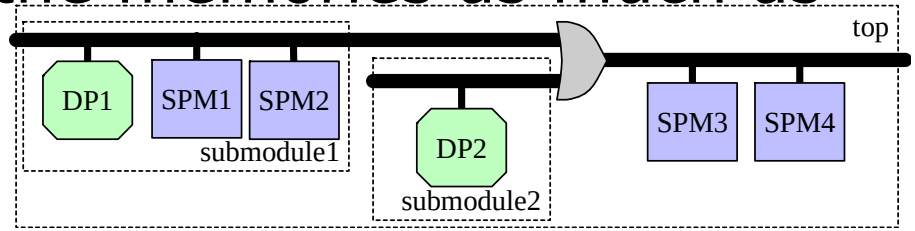




- Possibility of creating long combinational paths with many memory components connected in chain
  - **Dominator-based analysis** and **bus merging** (OR gates) for localizing the memories as much as



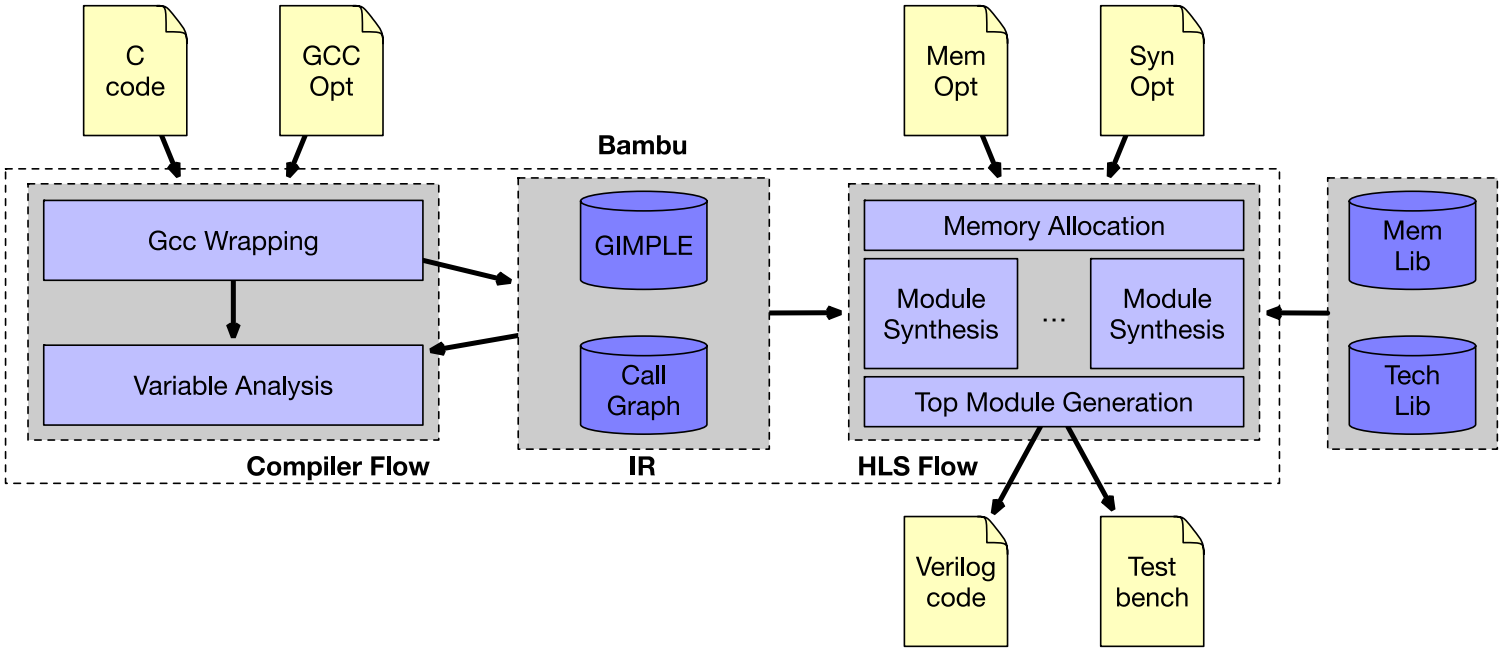
Daisy chain



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

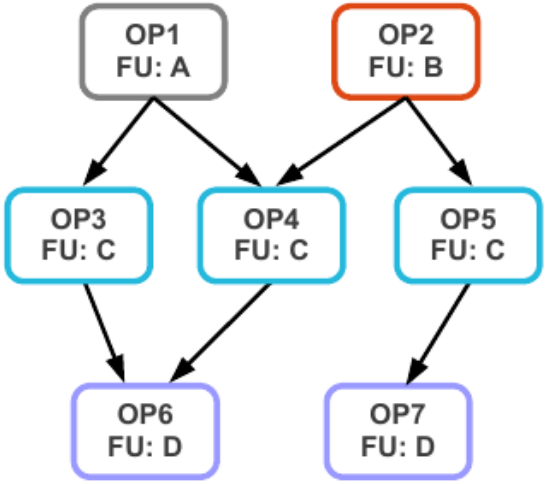
Parallel controller – dynamic scheduling

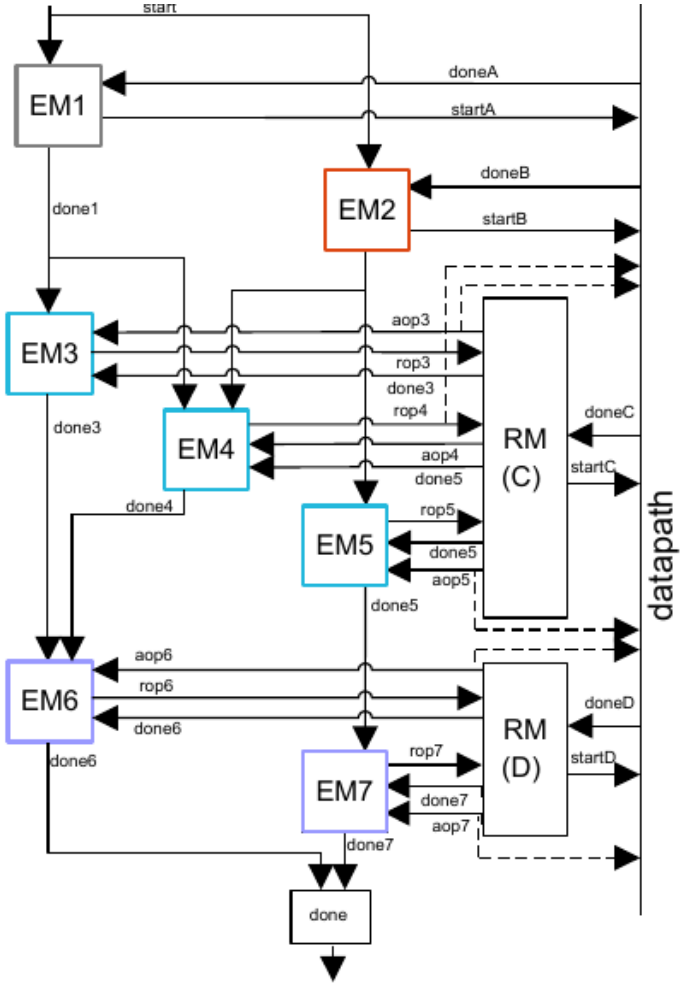
- Design of hardware components supporting failure detection
- Definition of a Parallel Controller (PC) architecture reordering the instruction execution given the current set of working components
- Extension of PandA HLS framework for self-adapting critical applications

Parallel Controller Design

- Set of communicating control elements, called Execution Managers (EM)
- Each EM establishes when an operation/task can start at runtime
  - Dynamic execution paradigm
  - Dedicated hardware for checking:
    - Satisfaction of dependence constraints
    - Resource availability
    - Failure detection
- Natural support for variable latency operations/tasks or for different number of resources available
  - ASAP executions also in these settings

Parallel Controller Architecture





Why a novel controller design?

- Complexity of the proposed controller increases linearly with the number of operations
  - **Regardless** to the operations latency!
- Allows **concurrent** execution of unbounded operations
  - Speculative operations
  - Memory accesses
  - Function calls
  - Hardware Ips
- Allow **dynamic reordering** of operations
  - Support variable latency operations
- Allow **self-adaptiveness**
  - Supporting different number of working components
- Complexity for managing n concurrent unbounded operations
  - proposed architecture: O(n) control elements
  - FSM controller: O(2^n) states and transitions