FROM RESEARCH TO INDUSTRY

cea tech

Strengths and weaknesses of LLVM as a code generator for an application-specific processor

Ivan Llopard

ivan.llopard@cea.fr

CEA/LIALP

April 4, 2013

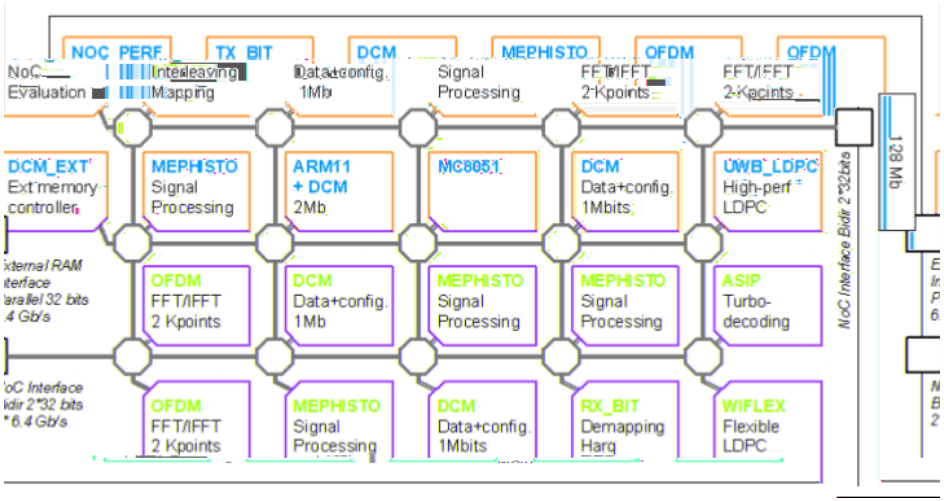
www.cea.fr

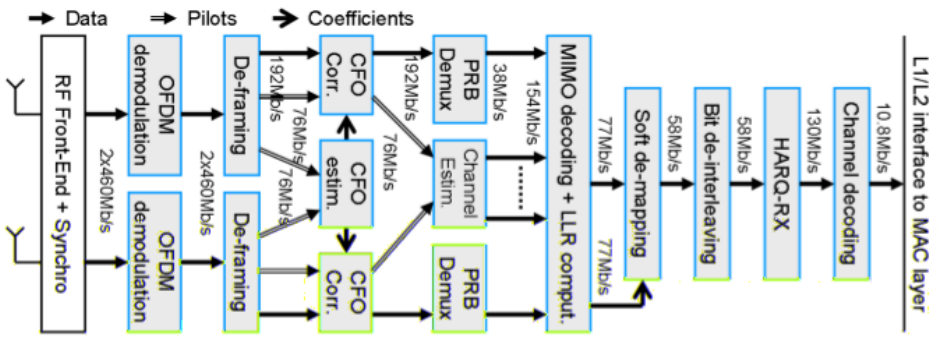
leti & list

Outline

Context

- Toolchain development for an application-specific processor, *Mephisto*
- Integrated in a Network on Chip architecture for digital baseband processing
  - 18 high-performance ASICs and 5 Mephistos
- Software Defined Radio (SDR) system
  - Implementation of Long Time Evoluation (LTE) specification
  - Data flow representation of the application





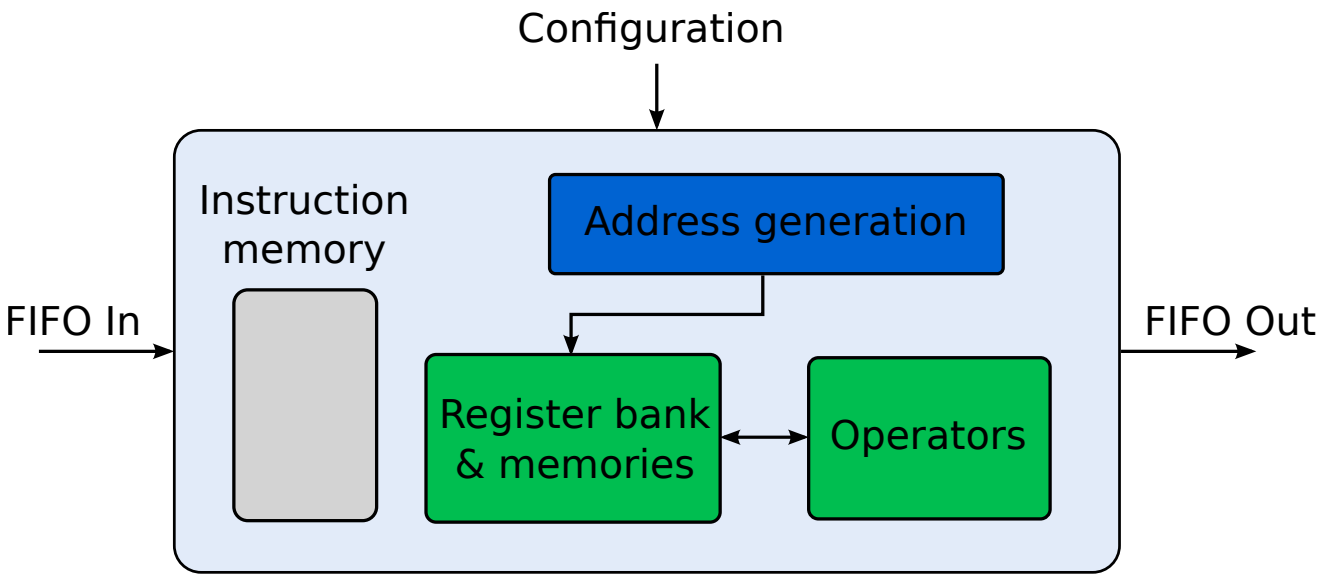
Outline

Mephisto

- Orthogonal VLIW architecture
- Low power consumption and high throughput
  - 50mW @ 3,2GOps
- Instruction 265 bits wide
  - Compacted by a higher level instruction cache (64 bits)
- Designed to handle the arithmetic of complex numbers (1 complex operation per cycle)
- No instruction decoder (fine grain control over the resources)
- Accumulation-based operations with saturated fixed point computations
- Indexed register addressing
- Clustered register files
- Address generation support (limited communication with data registers)

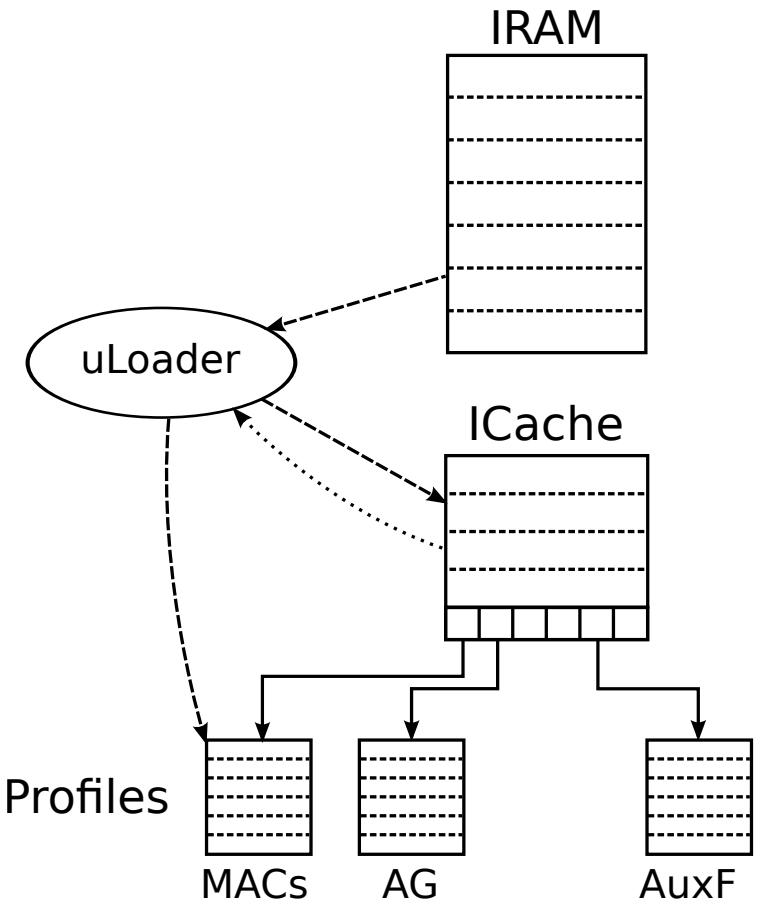
Overview of Mephisto

- The NoC controller loads the main program into Mephisto's memory and enable its execution
- Consume and produce data through input and output FIFOs



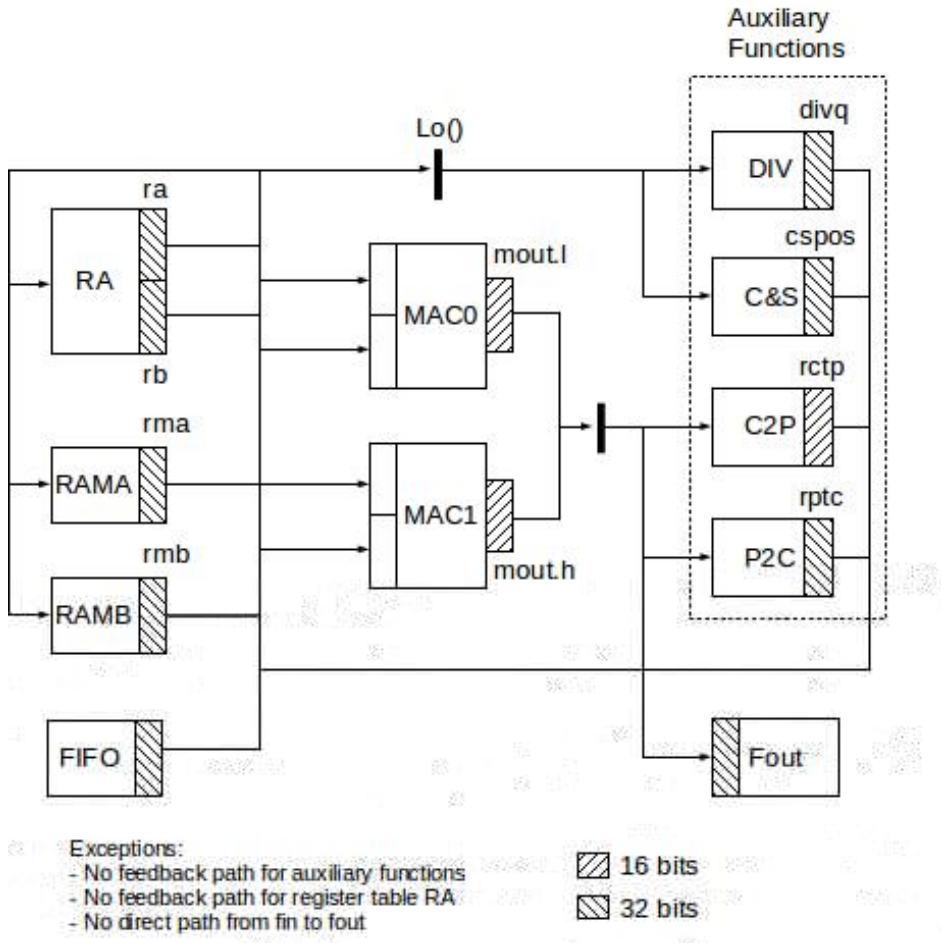
Mephisto: I-Path

- 64 bits Instruction RAM
  - Instruction as an array of indexes (bundle)
  - Address sequences for the address generators
- Indexed profile tables
- uLoader
  - Cache initialization
  - Update cache or profile entries
- Profile tables (265 bits)
  - MAC, FIFO, Auxiliary operators, Address generators, etc.
- Delayed indexation
- Hardware loop support
- No call instructions



Mephisto: Data-Path

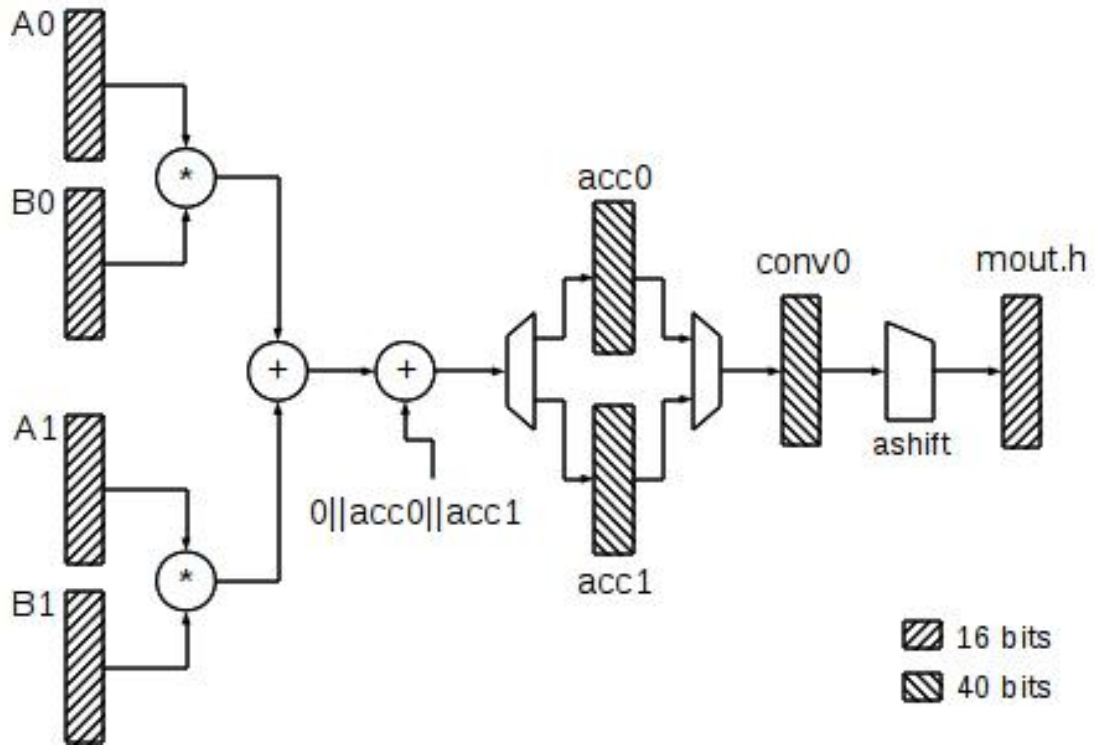
- Output register for every functional unit
- Multiply-and-Accumulate operators (MAC)
- Data
  - 1 Indexed register file (RA) with 2 output ports: 2x16bits
  - 2 Data RAMs with a minimum addressable unit of 4 bytes
- Input and output FIFOs
- Auxiliary functions
  - 2 CORDIC implementations
  - 1 Divisor
  - 1 Compare & Select



Outline

MAC functional structure

- Five-stage pipeline
- Positive or negative accumulations
- 40-bits Accumulators
- Arithmetical right shift and 16/31 bits saturation



MAC: Complex operation

- The MAC is able to perform *half* complex operations. Combining them, it can perform one complex operation per cycle.
- Examples
  - "Half" complex multiplication

```
load ra, [$1] ; load first output port
load rb, [$2] ; load second output port
; compute acc0 = Im{ra} × Re{rb} + Re{ra} × Im{rb}
mula acc0, ra.h, rb.l, ra.l, rb.h
asrsat16 mout.h, acc0, 10 ; saturate and shift
store mout.h, R:[$0] ; store to the register array
```

  - Accumulation

```
; compute acc0 += Im{ra} × Re{rb} + Re{ra} × Im{rb}
mula.add acc0, ra.h, rb.l, ra.l, rb.h
```

Integration of Mephisto in LLVM

- General rule: keep a reasonable amount of intrinsic functions

Integration of Mephisto in LLVM

- General rule: keep a reasonable amount of intrinsic functions
- Multiple memory spaces
  - Address space specification is supported (e.g. global-shared in PTX, fs-gs in X86, etc.)

Integration of Mephisto in LLVM

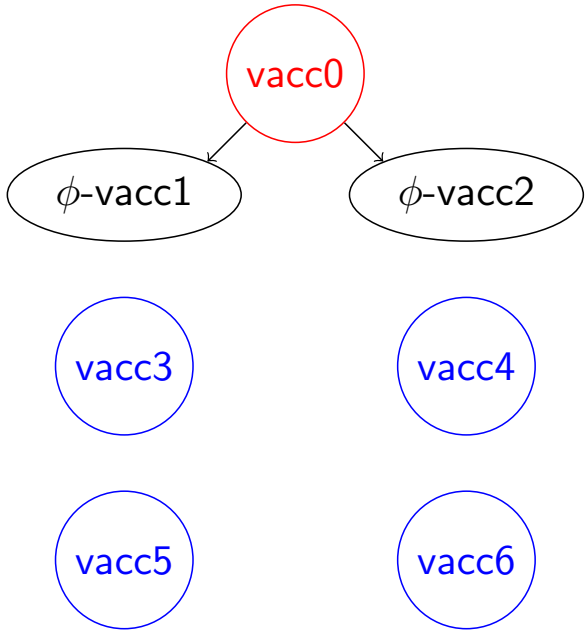
- General rule: keep a reasonable amount of intrinsic functions
- Multiple memory spaces
  - Address space specification is supported (e.g. global-shared in PTX, fs-gs in X86, etc.)
- Register type
  - *_Complex* is an aggregate
  - Built-in vector types: Makes pattern matching easier

Integration of Mephisto in LLVM

- General rule: keep a reasonable amount of intrinsic functions
- Multiple memory spaces
  - Address space specification is supported (e.g. global-shared in PTX, fs-gs in X86, etc.)
- Register type
  - *_Complex* is an aggregate
  - Built-in vector types: Makes pattern matching easier
- Non-spillable accumulators: cannot recover its original value because of saturation semantics
- Intra and inter-MAC accumulator copies are illegal
- Additions cannot be systematically promoted

Code generation for MAC

- MAC clustering
  - Consider both MACs as being two different clusters with its own register bank (accumulators)
  - The DAG is composed only by acc operations and ϕ-nodes
  - Assign a cluster to each operation
  - Instruction cloner and coalescer need to work together to avoid inter and intra-cluster copies

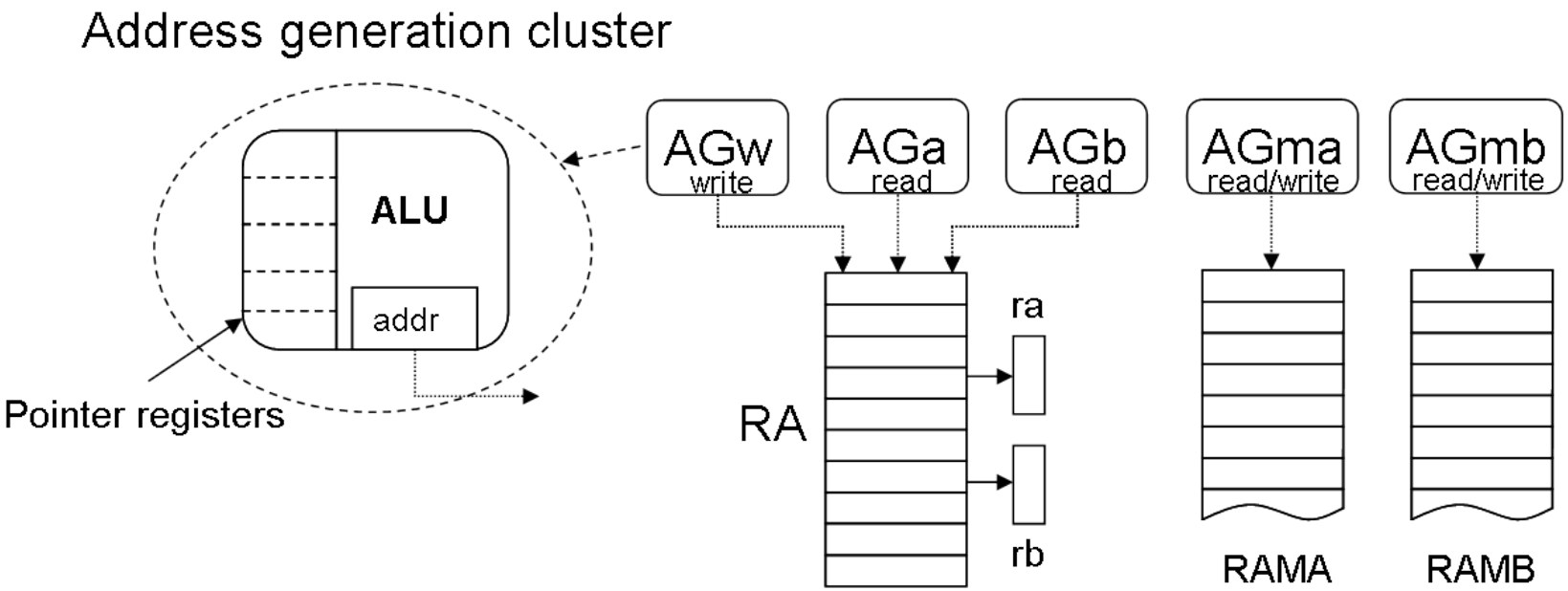


- Accumulators cannot be stored into main memory: memory promotion
- Spilling in register allocation: Force rematerialization
- Lack of saturation semantics: provide intrinsic functions

Outline

Address generation system

- Five address generators (AG) containing a bank of 8 pointer registers each one
- Support for modulo additions
- Static sequence of addresses
- Direct and indirect addressing modes
- The executable memory space is not addressable
- Pointer registers cannot be stored into memory (non-spillable)
- No communication path between AGs



Address generators in LLVM

- Considering a array mapped into the register file space, address computations for loads and stores cannot not be data related

```
reg[index] = reg[index] + reg[index-1];
index++;
```

Address generators in LLVM

- Considering a array mapped into the register file space, address computations for loads and stores cannot not be data related

```
reg[index] = reg[index] + reg[index-1];
index++;
```

- LLVM simplifies similar induction variables

```
reg[new_index] = reg[index] + reg[index-1];
new_index++; index++;
```

Address generators in LLVM

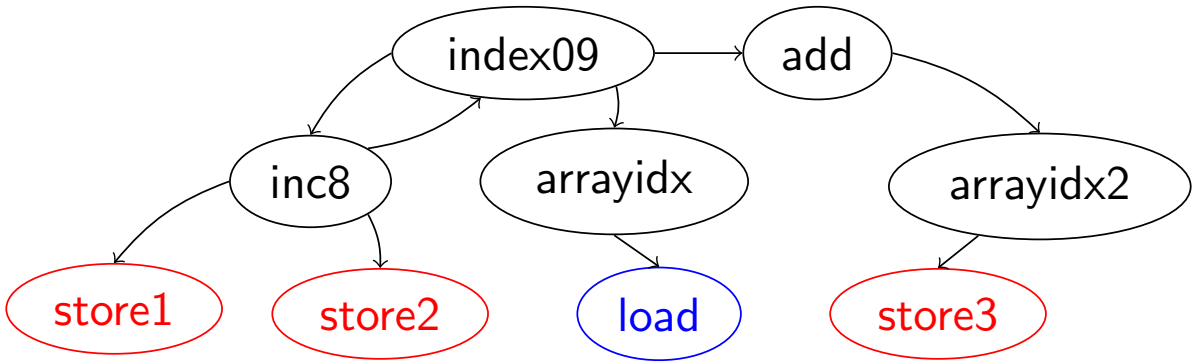
- Example

```
for (int index=0; index<16; index++) {
  reg[index] = readf();
  reg[index+16] = readf();
  reg[index] = reg[index] + reg[index + 16]
}
```

Address generators in LLVM

- Example

```
for (int index=0; index<16; index++) {
  reg[index] = readf();
  reg[index+16] = readf();
  reg[index] = reg[index] + reg[index + 16]
}
```

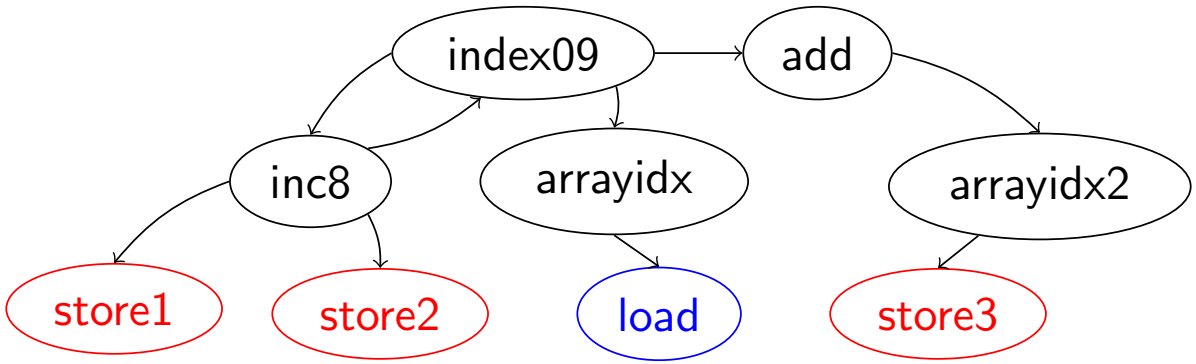


- Leads to instruction cloning as in the MAC clusterization solution
  - Impossible if it finds instruction with unknown side-effects

Address generators in LLVM

- Example

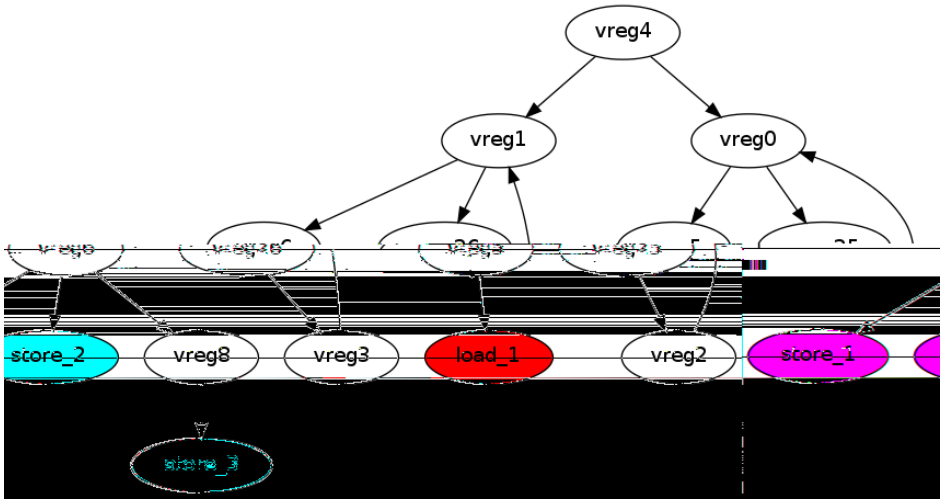
```
for (int index=0; index<16; index++) {
  reg[index] = readf();
  reg[index+16] = readf();
  reg[index] = reg[index] + reg[index + 16]
}
```



- Leads to instruction cloning as in the MAC clusterization solution
  - Impossible if it finds instruction with unknown side-effects
- Different load instructions following the output port they work on!
- Output port reutilization versus register pointer clusterization

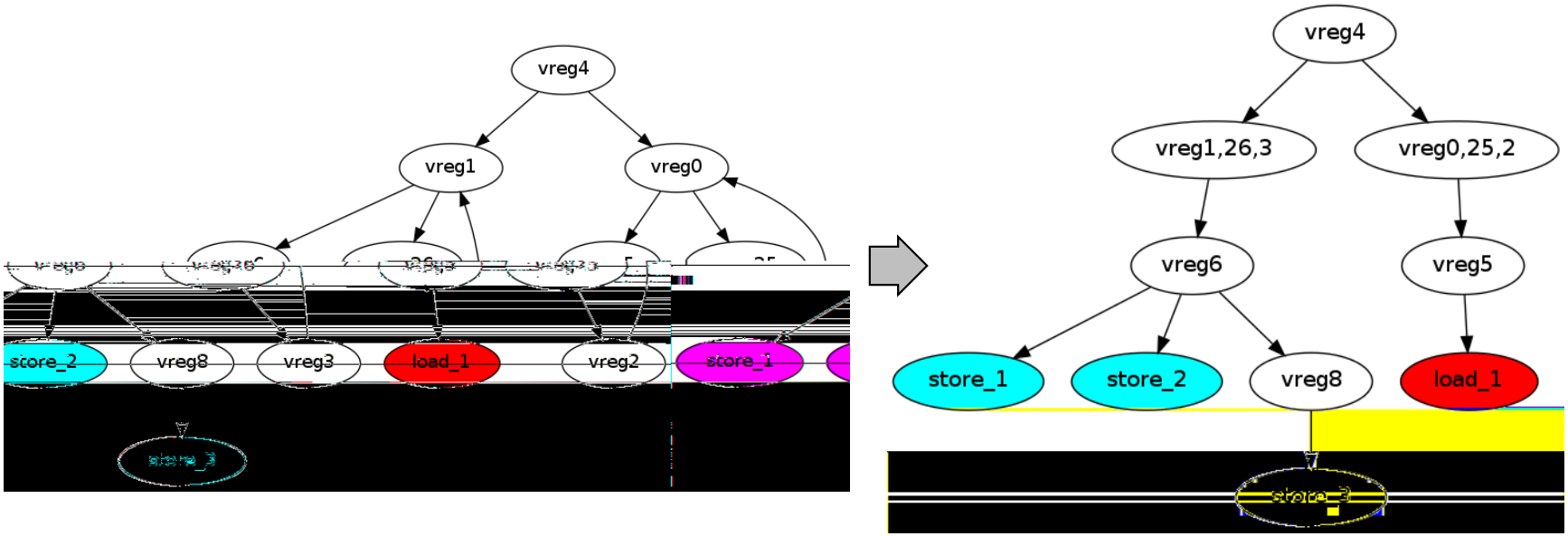
Strategies for instruction cloning

1. Find a path relating two conflicting nodes and clone it
2. Find the SCCs of the graph, reduce them (DAG) and find the meet of two conflicting nodes to start cloning from there



Strategies for instruction cloning

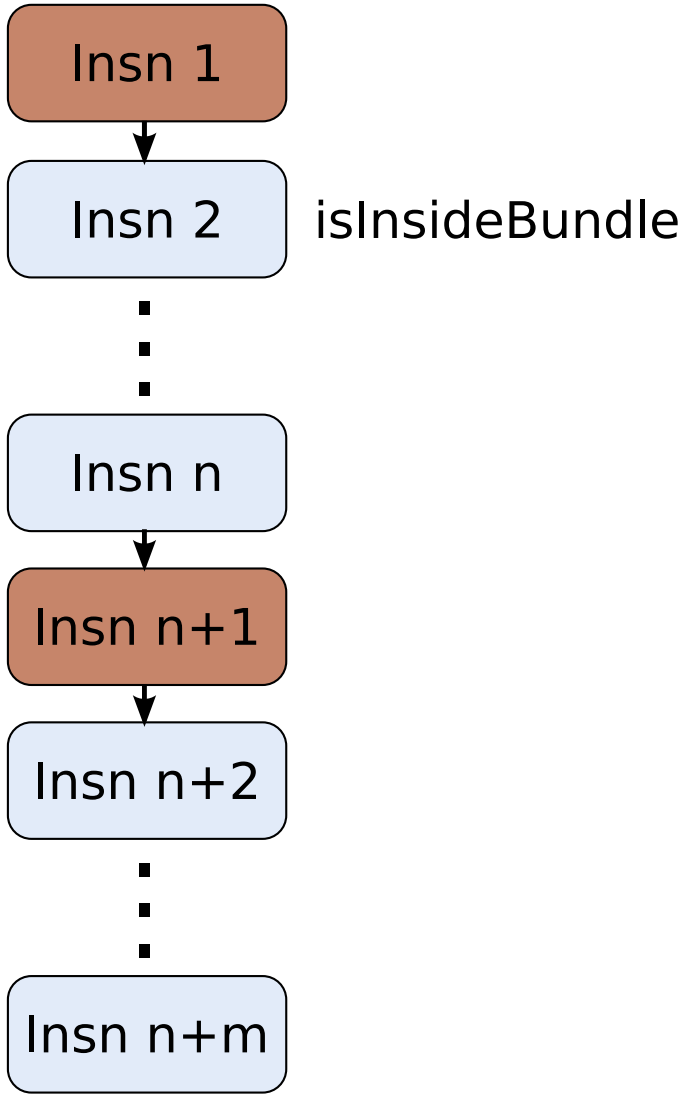
1. Find a path relating two conflicting nodes and clone it
2. Find the SCCs of the graph, reduce them (DAG) and find the meet of two conflicting nodes to start cloning from there



Outline

Instruction scheduling and bundling

- Since version 3.1, LLVM has a representation for the instruction bundle
- Transparent integration
- New iterator to get instructions inside bundles
- New flag to mark bundled instructions *isInsideBundle*



Instruction scheduling and bundling

- Live intervals are computed taking bundles into account

```
%vreg1 = opA %vreg0
%vreg3 = opB %vreg2<kill>
```

- No live interval interference between *%vreg1* and *%vreg2*
- Not all back-end passes are aware of bundles
- Finalized bundles
  - Replace bundle header for the special instruction opcode BUNDLE
  - Move instruction flags from inside the bundle to the header

Instruction bundles in Mephisto

- Instruction itineraries: Pipeline specification of operators (resource usage and latencies)
- Automatic generation of a DFA to check for valid instruction packets

Instruction bundles in Mephisto

- Instruction itineraries: Pipeline specification of operators (resource usage and latencies)
- Automatic generation of a DFA to check for valid instruction packets
- Overlapped instructions. Bundled iff the constant field is the same in both instructions

```
load ra, [$1]
add agra1, agra0, $1
```

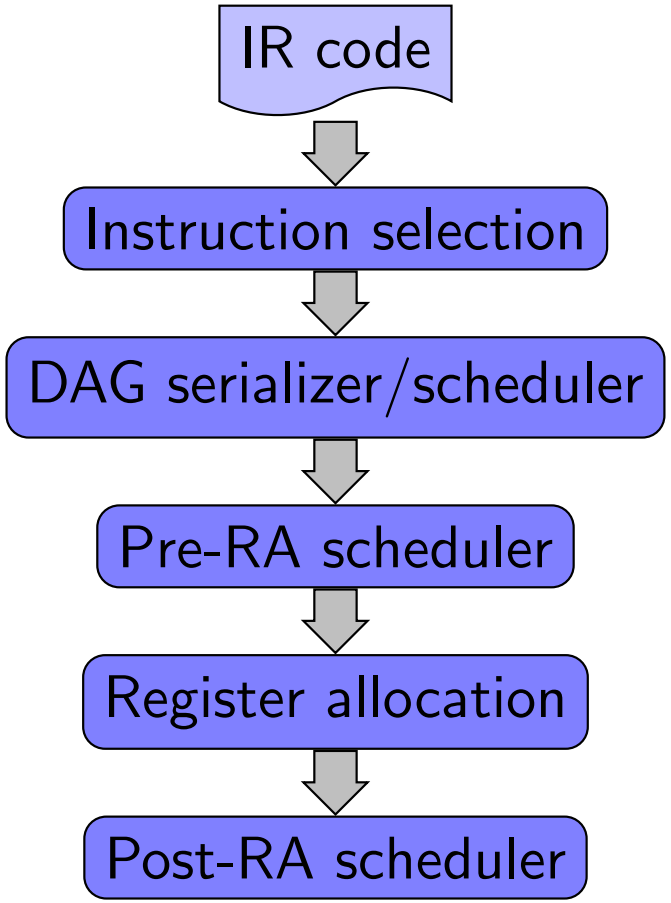
- Parallel post-increment. Bundled iff the same address register (*agra0*) is used in both instructions

```
load ra, [agra0]
add agra0, agra0, $1
```

- Usage of multiple resources at the first stage
- DFA is not enough!

Instruction Itineraries

- Non-interlocked processor
- Resource aware list scheduler (VLIW scheduler in LLVM)
- Instruction expansions after DAG serializer/scheduler
- Pre-RA scheduler for coarse grain scheduling
  - Shorten liveness of accumulators in basic block scopes
- Post-RA scheduler for fine grain scheduling
  - Fix data and resource hazards and produce final bundles



Outline

Data memory granularity

- The minimum addressable unit in Mephisto is 4 bytes (complex)
- Doesn't match the general assumption in LLVM: Byte-addressable machines

Data memory granularity

- The minimum addressable unit in Mephisto is 4 bytes (complex)
- Doesn't match the general assumption in LLVM: Byte-addressable machines
- The IR provides a good abstraction for address calculations: *getelementptr*

```
%arrayidx = getelementptr inbounds [16 x <2 x i16>]* %
    pilote, i16 0, i16 %index
```

- GEP lowering in SDAG construction for instruction selection

Data memory granularity

- The minimum addressable unit in Mephisto is 4 bytes (complex)
- Doesn't match the general assumption in LLVM: Byte-addressable machines
- The IR provides a good abstraction for address calculations: *getelementptr*

```
%arrayidx = getelementptr inbounds [16 x <2 x i16>]* %
    pilote, i16 0, i16 %index
```

- GEP lowering in SDAG construction for instruction selection
- This abstraction may be broken at different levels
  - Illegal casts coming down from C code (bitcasts)

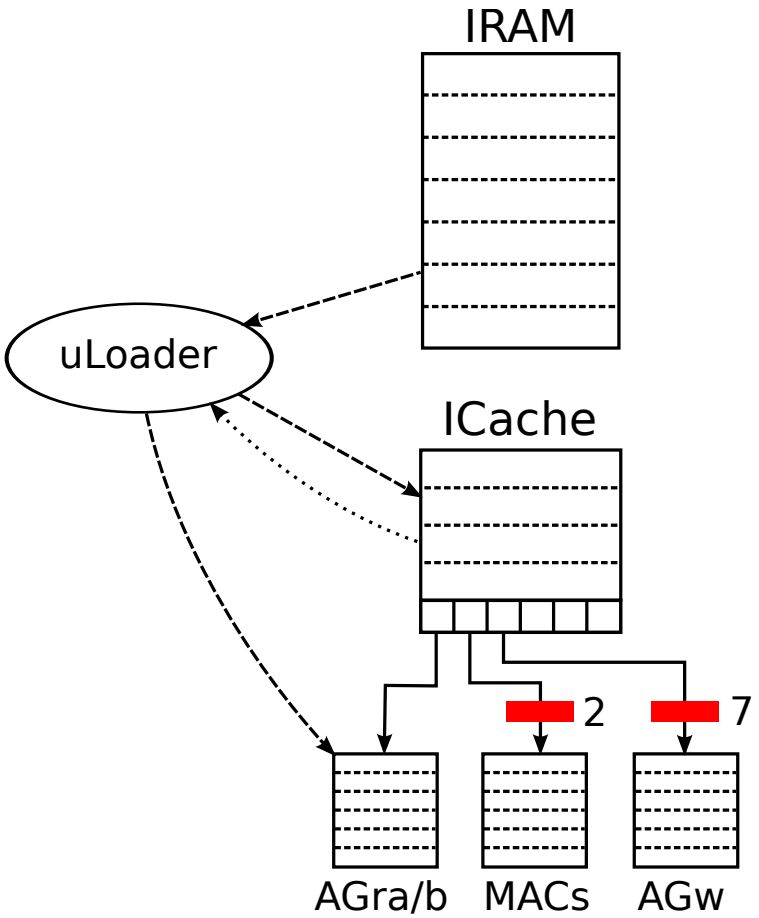
```
int a = 10;
char *ina = (char *)&a;
```

  - Optimization passes (Loop Strength Reduction)

Outline

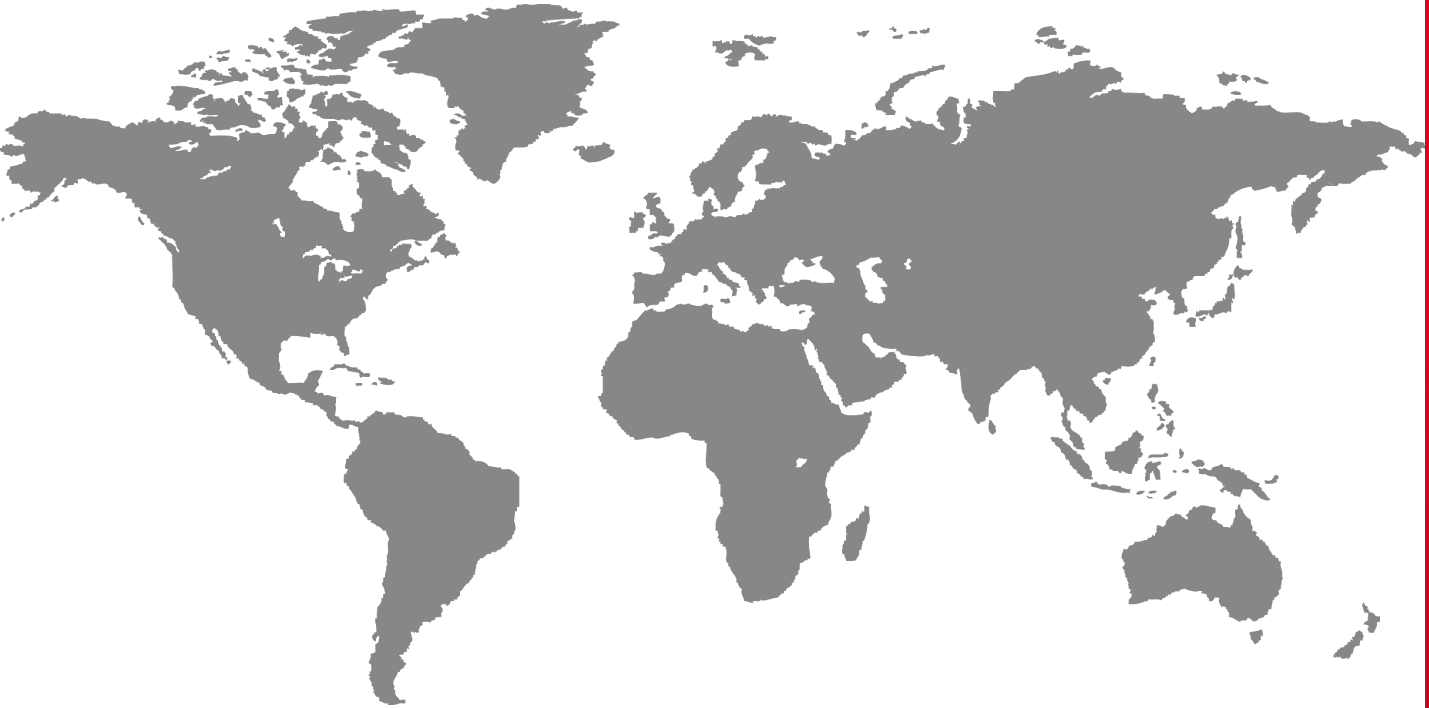
Binary code compaction

- Delayed execution of slots in the instruction packet
- Binary compaction for an efficient cache utilization
- Allow bundling of anti and *true* data dependencies
- Hazard recognizer complexity
- Factorization of profile entries
- In-advance page replacement



Conclusions

- + Built-in representation for instruction bundles
- + Automatic validity checks on bundles
- + Memory space specification
- - Support for saturation semantics and fixed point computations
- - Word addressable machines
- - Data-flow transformations to split calculations
- Translation to accumulation form of arithmetic operations?
- Scheduling of delayed bundles?
- Binary code compaction strategies (cache aware)?



Thanks!

leti
Centre de Grenoble
17 rue des Martyrs
38054 Grenoble Cedex

list
Centre de Saclay
Nano-Innov PC 172
91191 Gif sur Yvette Cedex

Programming interface

- Additional semantics for
  - Saturation
  - Input & output FIFO
- Complex type: LLVM built-in vector
- Exposed accumulators: *long long*

```
int2 D1, D2, R1;
long long acc0 = 0, acc1 = 0;
for (int i=0; i<400; i++) {
  D1 = readf();
  D2 = readf();
  // conj(D1) * D2
  acc0 += D1.x * D2.x + D1.y * D2.y;
  acc1 += D1.x * D2.y + D1.y * D2.x;
}
R1.x = asrsat(acc0, 18);
R1.y = asrsat(acc1, 18);
writef(R1);
```

Hardware loops

- It hides loop counter evolution and boundary checks
- It depends on target caractheristics
  - Up to 4 nesting levels
  - Cannot exit the loop before termination
- Target-independent hardware loop builder
- High-level (IR) trip count computation using induction variable analysis (ScalarEvolution in LLVM)
  - New loop intrinsics: enterloop/endloop

```
entry:
  call void @llvm.meph.enter.loop(i16 199)
  br label %for.body
for.body:
  ...
  %1 = call i16 @llvm.meph.end.loop()
  %2 = icmp slt i16 %1, 0
  br i1 %2, label %for.end, label %for.body
```