LANGUAGES AND COMPILERS FOR EXTREMELY RESOURCE CONSTRAINED PROCESSORS

LangDev 2022

Luca De Santis

Introduction

Setting a scenario :

- Integrated circuits with embedded control of physical quantities
- Control algorithms are complex
- Many hard constraints
  - Cost/Area
  - Speed
  - Low Power
  - Time-to-market

Constraints

Metric	Requirements	Best Choice
Very Small Silicon Foortprint	Every instruction has a cost in terms of area	Processor
Tight Real Time Control	Every instruction has a time cost Accurate timing control of Physical Processes	Dedicated HW
Very aggressive Time to market	Reuse, Robustness, Bug-free silicon delivery, Verification cost	Processor (decoupling HW verification from SW verification)
Ultra Low Power	Every instruction has a power cost	Dedicated HW

In this context we generally think to the «processor» not as a «component», but as a «design methodology» , aiming to minimization of silicon area and maximization of reuse and robustness, paying in performances and power.

The fundamental questions

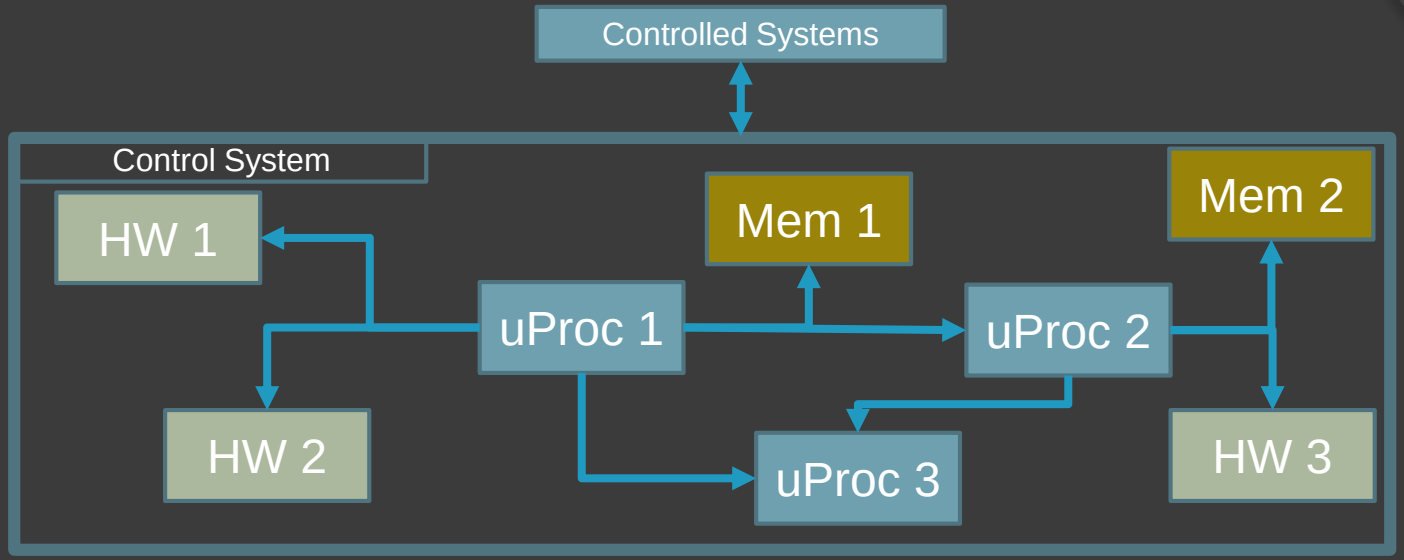
- Why is there a need for **custom processors** ?
  - Control algorithms are enough complex so that Finite State Machine based design isn’t viable
  - Commercial processors are redundant and area consuming, instruction set must be tailored on the control algorithms
- Why is there a need for **custom languages and compilers** for custom processors ?
  - Control algorithms are enough complex so that assembly programming isn’t viable
  - Commercial compilers are redundant
  - Optimization must be done on the full software/hardware stack

Topics

- Architecture-Aware Programming
- Instruction replacement
- Compiler-in-the-loop concept
- Other specific topics

Architecture Aware Programming

Today Digital Architectures are something like:

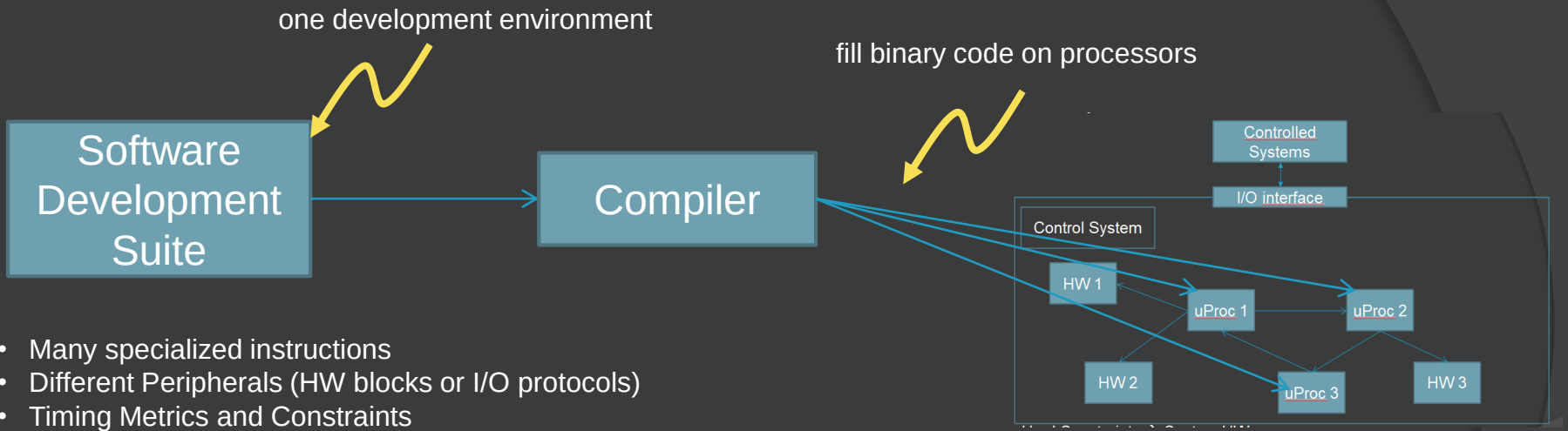


A set of communicating and specialized processors plus a set of dedicated HW and memories

- Classical heap/stack model common to many programming languages doesn't fit
- Memories have different sizes/bit-width
- Communication overhead can't be tolerated

Programmer must know about architecture to write efficient code
Compiler must know about architecture to emit efficient bit-code

Programmer's point of view



- Many specialized instructions
- Different Peripherals (HW blocks or I/O protocols)
- Timing Metrics and Constraints
- Code size efficiency (execution time vs. total code amount)
- Abstract representation of architecture (address boundaries, data types, data movements)
- No OS support (peripheral management is quite complex)

Abstraction has a value but every abstraction has a cost
Customization makes abstraction efficient
«Efficient customized abstraction» has a higher value

Typing

Type system is simple but there is a problem: to minimize area and improve timing, data are stored in memories with different size and access policies

```
def x as reg16
def y as ram8
set x[15:8] = y
```

Data movement constraints

Storage media	size	access	IR	RAM1	RAM2	PROM	Act
Internal registers	16	RD/WR	yes	yes	yes	no	yes
RAM1	8	RD/WR	yes	no	no	no	yes
RAM2	12	RD/WR	yes	no	no	no	no
PROM	10	Read	yes	no	no	no	no
Action registers	8	Write	no	no	no	no	no

Compiler must check legality of data movement

Open problem: optimal allocation of storage resources

Instruction Replacement

A complex feature: Instruction replacement

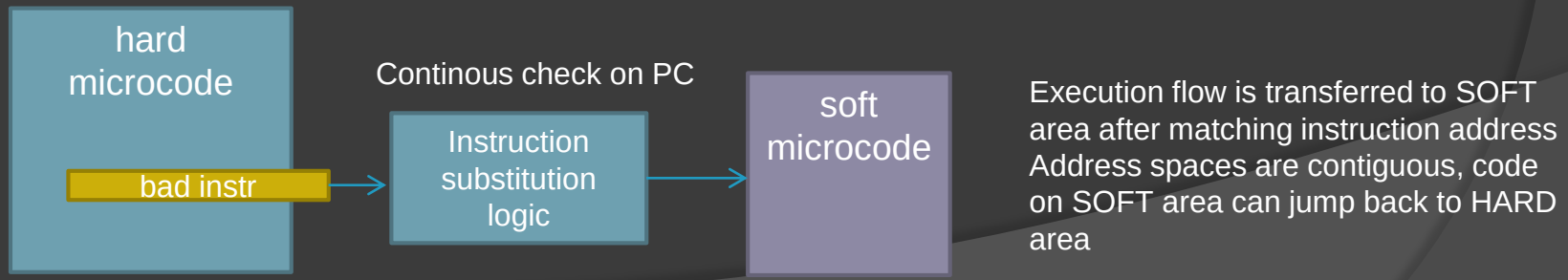
Problem :

- Software is hard-coded on silicon
- After debug some instructions must be fixed

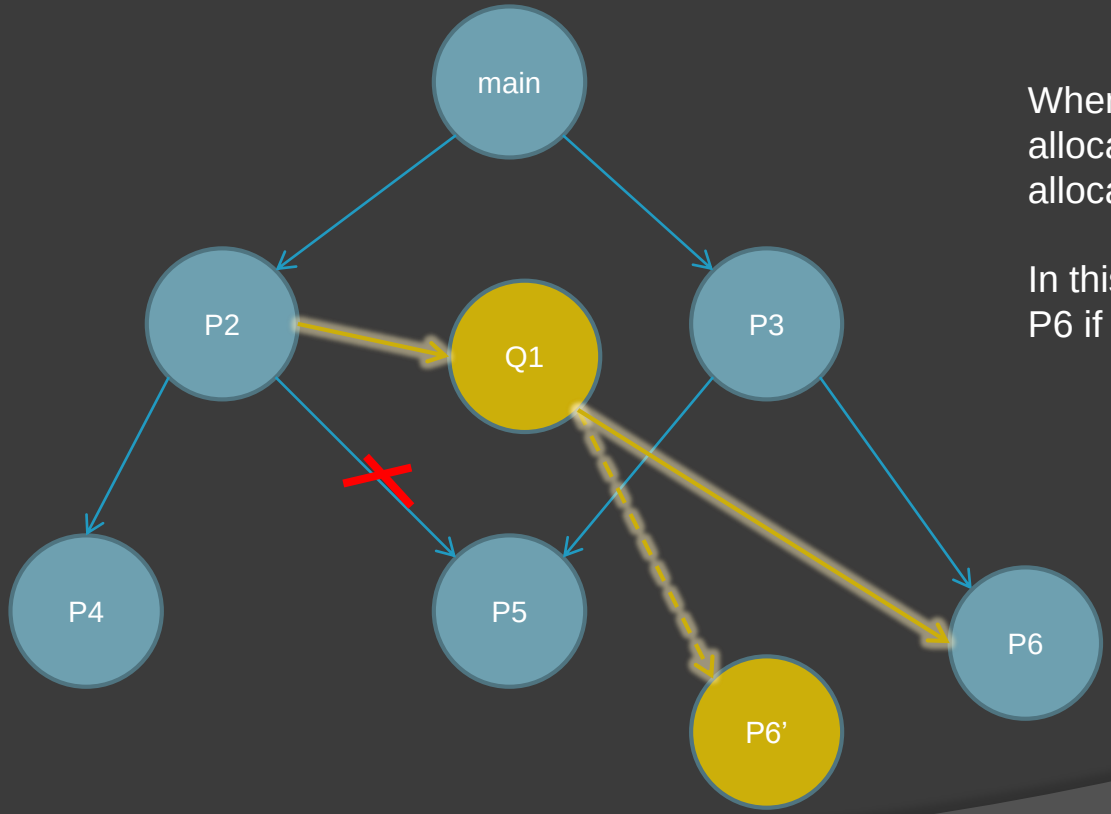
Solution: part of the code is in a volatile area filled at system's power-up

Problem for the compiler : guarantee registers coherence

- In other words the Procedure DAG can change, but most part of information are hard-coded, so the new code must be compiled so that it doesn't break previous register allocation strategy



Registers coherence after fix



When allocating register for Q1, allocator must take care of existing allocations on HARD micro-code

In this example, compiler must duplicate P6 if P2 and P6 share registers

Examples

Single row replacement

```
…
set x = 0 ;
replace( set x = 3 );
…
```

Procedure call replacement

```
…
call P1 ;
replace( call P2 );
…
```

Adding instructions

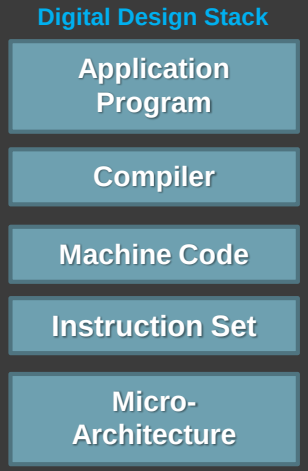
```
…
set x = 0 ;
replace(
        set y = 0 ;
        call P1 ;
        set x = 0 ;
)
…
```

Jump to SOFT area

Jump back to HARD area

Compiler in the loop

Compiler-in-the-loop concept



Problem:

- Let's suppose we want to improve a digital architecture (HW+SW) in terms of area/power/speed metrics
- We want to reuse SW as much as possible
- Any change in the architecture/instruction-set means to change the compiler and the program
- This is time-consuming and bug risky

Compiler-in-the-loop

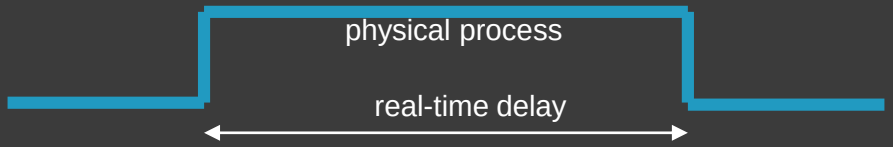
Possible solutions:

- Virtual machine – too slow for some applications
- Intermediate language – sub-optimal because compiling to IL makes us lose some useful information for real-time control
- Parametric/Generic compiler, so that compiler is in the trial loop (something like LLVM as in Synopsys ASIP Designer)

Open problem: full stack optimization with timing info
Possible path: graph rewriting techniques, need of a language for common description of HW and SW.

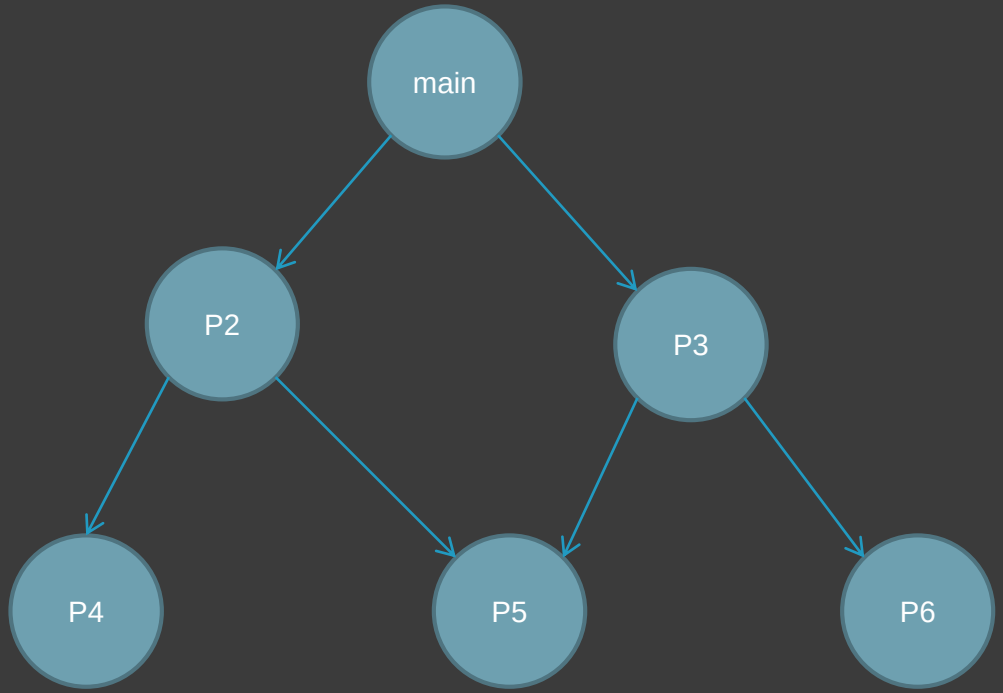
Other specific topics

Real-time control = physical process control



- Physical process control gives an interesting opportunity: use delay time to «hide» computations and other processing specific stuffs
- This makes programmer comfortable to write «relaxed» code
- But specifics for that changes over time
- Suddenly, that code becomes dramatically fatal, because violates RT constraints
- No automatic management of this issue today: research opportunity ? **Real-time-aware compiler** ? Tagging code with timing info ?

Dealing without a stack



Traverse Procedure Dag

Each Procedure P has a register allocated for return address, no local variables, no parameters passing

No recursion (direct or indirect) allowed

Traverse DAG, allocating one register for return address to each procedure.

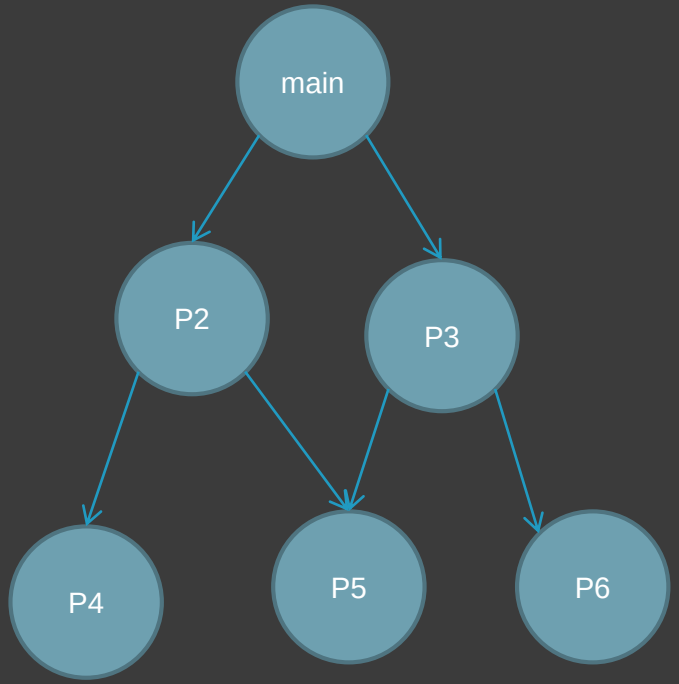
Before allocating a register, analyzer has to traverse hierarchy.

When going back to father, free registers

Consistency condition : In any path from root to leaves, each register must be allocated only once

In the example, when allocating a register for P3, we have to check previous allocation for P5, possibly used when traversing P2

Dealing with only a stack for return addresses



Return addresses are managed automatically by HW

As in the previous example, each procedure P has some registers allocated for parameters passing and return value.
No recursion (direct or indirect) allowed

Return value is a register allocated on the caller.
The callee access it by Pname.varname
Procedures like P5 must use global variables

Problem: fast saturation of registers

A new type of local variable : programmer can use it only before a new procedure call; in other words existence of variable is not guaranteed along the procedure call chains.

```
proc P3
int retval1
call P6
…
proc P6
P3.retval1 = 5
return
```

Simple but complicated type system and arithmetics

- No necessity of data structures
- Sometimes you have «hardware» pointers
- Signed + unsigned without cast
- Single bit operations/checks (more than C-like bitwise operations)
- Bit range selection X[6:4], right shift and left shift in the same clock cycle
- Saturation arithmetic, sometimes not aligned with word width
- Multiplication without a multiplier : x * 3 = x << 1 + x

Procedure call policies

- Call procedures on different processors
- Queued Calls
- «Inline» on the call
- Embedded return


```
…
set x = y
return
```

Opportunistically compiled as one instruction

```
…
call P
…
inline call P
…
```

fast vs. area-efficient code

Looking at the future…

- ASIC development has gone into integration of Verilog/VHDL IP blocks: «wild wiring» approach, huge cost of verification
- Verilog/VHDL abstraction is going into adopting high-level programming concepts (OO, FP..) but hardware designers are not so comfortable on those stuffs
- High-level synthesis is having a renewed boost in data-intensive applications on FPGA (hardware acceleration of mathematical algorithms): not well suited for control-intensive applications
- HW/SW co-design is focused on optimal partitioning of HW and SW tasks, but in this context processor architecture is an input data
- Processor synthesis is focused on instruction-set micro-details, not in the overall optimization, and it'is monopolized by Synopsys
- Compiler technology is going to multi-level optimization (MLIR), not well suited for highly constrained HW design

Dreaming…

A programming language for control algorithms that allows the definition of:

- Hardware boundaries and constraints
- Real-time constraints
- Power constraints

and a compiler that *suggests* the optimal architecture.

A bigger dream is that …

the compiler *synthesize* the optimal architecture.

Thanks for attention…

https://www.linkedin.com/in/luca-de-santis-42a87a5/
ldesantis@ymail.com