Enhancing hardware design through domain-specific language tools for higher abstraction

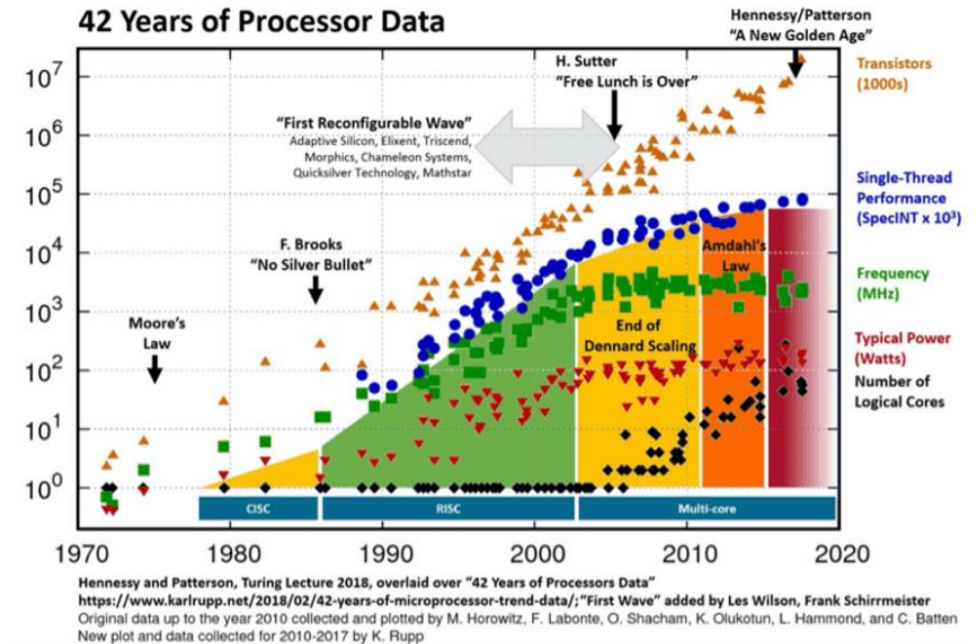
Luca De Santis / Micron Technology Italy

C8B2

Seville 17-19 October, 2024
https://langdevcon.org

The context

- For many decades, hardware design has been driven by the processor model: addressing problems at the software level and executing code on general-purpose hardware.
- With the end of Dennard scaling, the focus shifted toward multi-core and multi-processor architectures.
- To achieve better performance, reduce costs, and lower power consumption, we have now entered the era of **heterogeneous computing**. In this approach, complex digital systems are composed of a blend of standard processors, customized processors, storage media, complex state machines, reconfigurable hardware and ad-hoc accelerators.



Turing Lecture 2018

Innovations like domain-specific hardware, enhanced security, open instruction sets, and agile chip development will lead the way.

BY JOHN L. HENNESSY AND DAVID A. PATTERSON

A New Golden Age for Computer Architecture

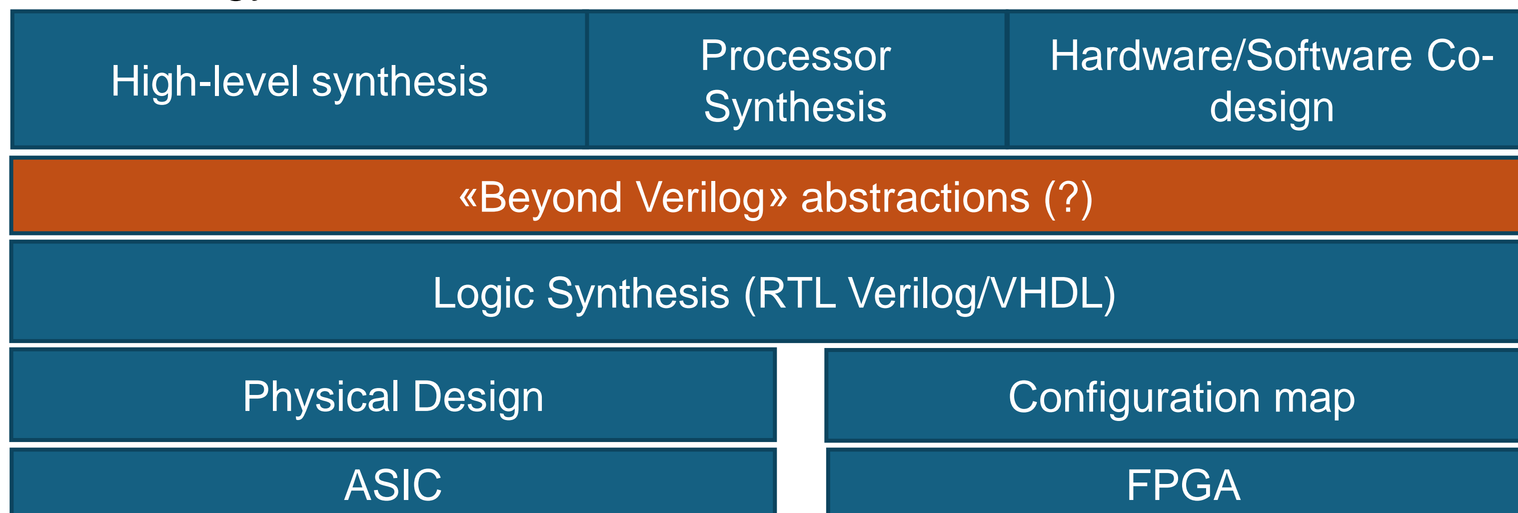
The problem

- Hardware design relies on the concepts of **Hardware Description Language** (HDL) and automatic synthesis.
- In the 80s, two major HDLs were introduced: **Verilog** and **VHDL**, and today all industrial-level chip development flows are based on these ones.
- However, as system complexity grew, the limitations of these older languages became evident. This highlighted the need to:
  - Increase the level of abstraction in hardware design.
  - Introduce “stronger typing for wires” in HDLs, preventing arbitrary connections between components. Some researchers have pointed out that hardware designers are now facing challenges comparable to the "goto" problem that software engineers encountered during the early development of programming languages.

Some solutions: the EDA landscape

Over the years, several attempts have been made to move beyond the digital abstraction in hardware design:

- **High-level synthesis**: Focused on algorithms described using programming languages. It has been highly successful in the DSP domain and data-intensive systems but is not as well suited for control-intensive systems.
- **Processor synthesis**: Centered on defining instruction sets. This approach involves expensive tools, and having the compiler integrated in the process adds significant value.
- **Hardware-software co-design and co-synthesis**: Relies on heuristics, it is not yet a fully developed or standardized methodology.



The real problem

«Beyond Verilog» abstractions are evolving toward incorporating software engineering concepts into hardware design, but:

- Hardware engineers generally do not have the same mind-set as software engineers.
- Hardware design requires strict adherence to fundamental constraints:
  - Area/performance/power trade-off
  - Fine tuning of timing (low level synchronizations, physical delays)
  - Strict control of «states», to avoid undesired hidden ones and deadlocks
  - Practical tools for definition of customized instruction sets and communication protocols
  - Readability of synthesized code for debug

A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity

Lenny Truong
Stanford University, USA
lenny@cs.stanford.edu

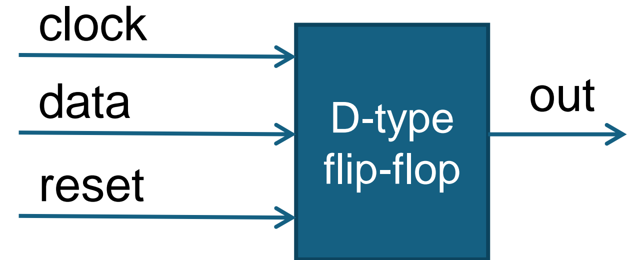
Pat Hanrahan
Stanford University, USA
hanrahan@cs.stanford.edu

3rd Summit on Advances in Programming Languages (SNAPL 2019)

The challenge

- There is a common feeling that the right way to face digital systems complexity is by increasing the level of abstraction of hardware description languages to stimulate a «**correct-by-design**» approach
- A common objection is that abstraction means losing details, but looking at the famous Dijkstra's citation: "**The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise**"
- **Can we adopt concepts and tools from the domain-specific language (DSL) field to meet this need?**
- Key point is: to capture what is the «**semantics**» of digital systems representations at any level of description

One example



- The simplest useful object in hardware design is the **D-type flip-flop**: it samples a signal when a clock event occurs; on the right is reported a description in synthesizable Verilog

```
always @ (posedge clock or
posedge reset)
          if (reset) out = 0 ;
          else out = data ;
```

- The problem here is that clock, reset and data can be everything; constraints on these signals are not defined at language level but on post-processing tools
- Here is how Chisel (one of the most promising languages aimed to increase the level of abstraction of hardware) defines a D-type flip-flop

```
class ExampleModule extends Module {
  val io = IO(new Bundle {
    val d   = Input(Bool())  // Data input
    val q   = Output(Bool()) // Output
  })

  // Define the register with reset behavior
  val reg = withReset(reset.asBool) {
    RegNext(io.d, init = false.B) // Initialize the register to 'false'
(reset state)
  }
  // Connect the output
  io.q := reg
}
```

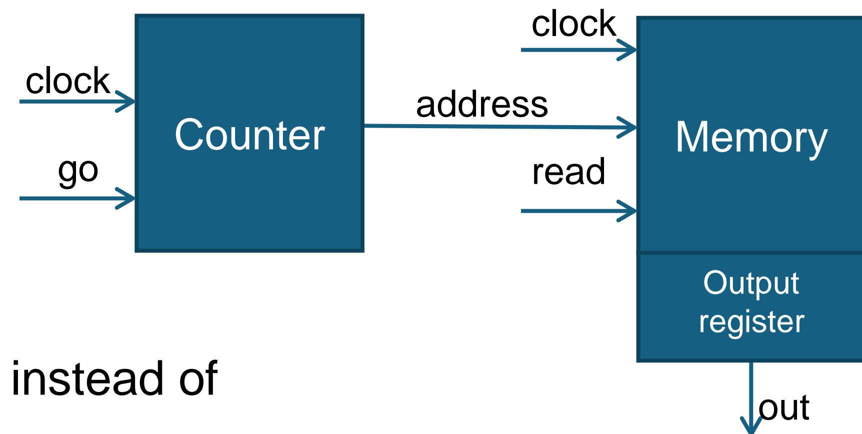
- Here is what a hardware designer would like to have: **explicit declaration** of clock and reset domains

```
Domain clock, reset  {

          Reg out = data {init(0)};

}
```

Another example

- A system composed of a memory and a counter that flushes data according to the value of an input signal «go»

```
Memory mem ( .addr(cnt),… ) ; // connect memory address to counter
always @ (posedge clock or posedge reset)
           if (reset) begin cnt = 0 ; read = 0 ; end
           else if (go) begin cnt = cnt+1 ; read = 1 ; end
           else begin cnt = cnt ; read = 0 ; end
```



- Here is what a hardware designer would like to have: **actions** instead of **wiring**

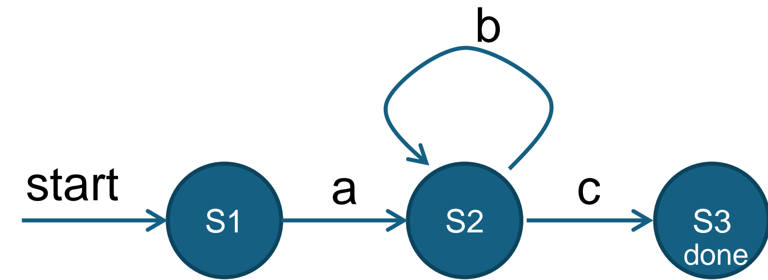
```
if go -> cnt.incr() , mem.read(cnt) ;
```

- This is an example of a very basic memory driver, made simple by introducing the concept of an «**action**» over a block, overtaking the practice of just «**wiring**» blocks

One more example

- **Finite state machine (FSM)** is one the fundamental concepts in hardware design. Defining real-world state machines can be highly challenging due to the need for exhaustive coverage of “state/input/next state/output“ tuples and the difficulty of identifying and managing errors.
- This is an example of a simple state machine: the classical exercise of a recognizer of strings composed of initial symbol ‘a’ followed by an arbitrary number of symbols ‘b’ and a final symbol ‘c’

```
if (state == idle)
          if (start) state = S1 ;
          else state = idle ;
else
          if (state == S1 & in=='a' ) state = S2 ;
          else if (state == S2 & in=='b' ) state = S2 ;
          else if (state == S2 & in=='c' )
                    begin
                              state = S3 ;
                              done = 1 ;
                    end
          else if (state == S3) state = S3 ;
          else state = error ;
```



- Here is what a hardware designer would like to have: **tables** instead of **code** with special symbols like ‘**any**’ and ‘**hold**’ ;

```
Table(state, in -> state, out) {
          idle, start -> S1 ;
          S1 , 'a' -> S2 ;
          S2 , 'b' -> hold ;
          S2 , 'c' -> S3 , done;
          S3 , any -> hold ;
          any, any -> error ;
}
```

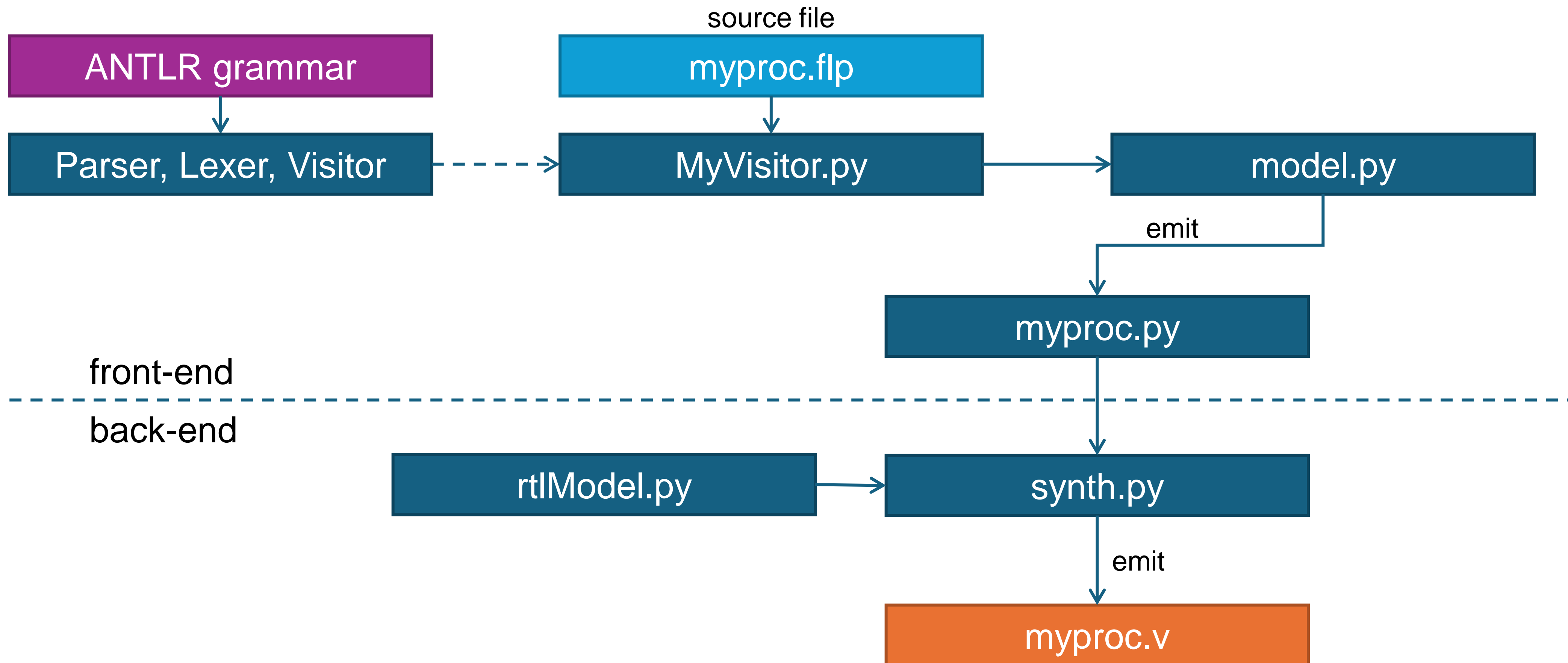
The proposal

- Explicit definition of clock and reset domains
  - Easier and cleaner definition of registers and sequential circuits
  - Automatic synchronization between domains
- Actions
  - Easier and cleaner definition of interfaces between some standard building blocks like memories, stacks and queues
  - High-level description of protocols
- Enhanced state transition tables
  - Compact definition of complex state machines and customized instruction sets
  - Easier error detection
  - Helps to avoid unwanted hidden states

Design Criteria for a language

- One file – One module (easy prototyping, forces engineers to be organized)
- Common HDL features:
  - Hierarchical instances, in/out ports, parameterization
  - Basic digital types (bit strings, bin/hex literals)
  - Typical digital operators (wires concatenation and slicing)
- Strong typing of wires, registers and memories
- Explicit clock and reset domains
- Clear differentiation between «stateless» vs. «stateful» objects
- Standard storage media (memories, stacks, queues…) and access protocols
- Multidimensional array access (for ML/AI accelerators)
- Enhanced transition tables with actions
- Iterators for indexed declarations

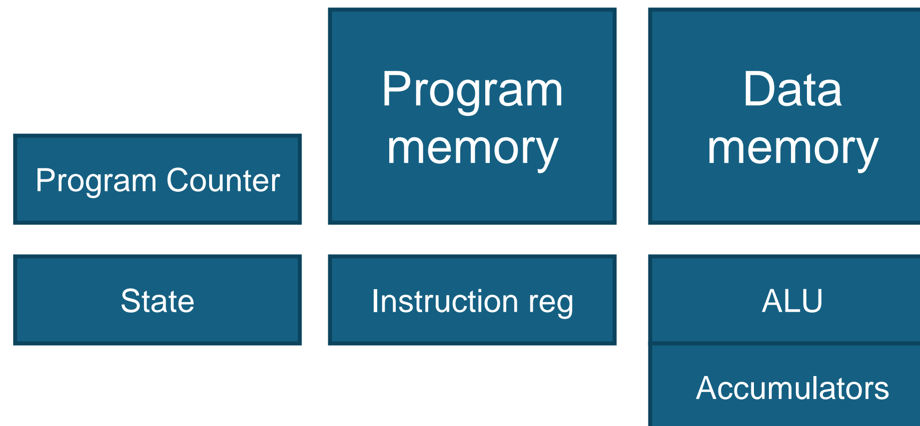
A possible flow (ANTLR+Python)



Typing (ANTLR grammar)

```
topItem:
      'Parameter' ID (sz = size)? ('=' e = expr)? ';'                # parDecl
    | 'Wire'      ID (sz = size)? (a = attrs)? ';'                   # wireDecl
    | 'Input'     ID (sz = size)? (a = attrs)? ';'                   # inpDecl
    | 'Clock'     ID (sz = size)? ';'                                # clkDecl
    | 'Reset'     ID (sz = size)? ';'                                # rstDecl
    | 'Index'     ID  ('=' e = expr)? ';'                            # iterDecl
    | 'Logic'     ID (sz = size)? (a = attrs)? ('=' e = expr)? ';'   # logicDecl
    | 'State'     ID (sz = size)? (a = attrs)? ('=' e = expr)? ';'   # stateDecl
    | 'Reg'       ID (sz = size)? (a = attrs)? ('=' e = expr)? ';'   # regDecl
    | 'Instance' n = ID (sz = size)? 'of' m = ID (msz = size)? (
        a = attrs
      )? (c = conns)? ';'                                            # instDecl
    | 'Action' ID ':' a = action ';'                                 # actDecl // TO
    | 'Action' ID ':' '{' alst += action (';' alst += action)? '}'   # multiActDecl
    | 'Table'  ID h = tableHead b = tableBody                        # tableDecl
    | 'Domain' c = ID ',' r = ID '{' (t += topItem)* '}'             # domDecl
    | 'assign' lhs = lhsexpr '=' rhs = expr ';'                      # assignment;
```

A simple processor as a first example



Instruction Set :

- NOP , HALT , JMP , JMPR, JMPZ, JMPNZ, IJMP, IJMPR, IJMPZ, IJMPNZ, ERR , LOAD, STORE, SETV,
- ADD, SUB, EQ, NEQ, LT, LTE, GT, GTE, AND, NAND, OR, NOR, XOR, XNOR, NOT

Source code entry

```
Parameter DW            =    8 ;
Parameter CODE_SIZE     = 1024 ;
Parameter CODE_WIDTH    =   21 ;
Parameter CODE_ADDR_SIZE =  10 ;
Parameter DATA_SIZE     =  128 ;
Parameter REG_SIZE      =   32 ;

Input go ;
Clock clk ;
Reset rst_ ;

Domain clk, rst_ {
    State state [4]  { enum(IDLE , EXE , FETCH, ERROR, DONE, EXE_ALU,
                            EXE_ALU_FLAG, STALL_LOAD, STALL_ALU, STALL_FLAG) } ;
    Reg   pc    [CODE_ADDR_SIZE] ;
    Reg   ir    [CODE_WIDTH] ;
    Reg   flag  [8] ;
    Reg   acc   [DW] ;
    Reg   acc2  [DW] ;
    Instance ram of mem [DW,DATA_SIZE] {init("ram_init.bin")};
    Instance rom of bank [CODE_WIDTH,CODE_SIZE]  {init("rom_code.bin")};
    Instance rf  of bank  [DW,REG_SIZE] ;
}
```

```
// combinationals
Logic icode [5] { enum(NOP  , HALT , JMP , JMPR, JMPZ, JMPNZ, IJMP, IJMPR, IJMPZ,
                       IJMPNZ, ERR , LOAD, STORE, SETV,
                       ADD, SUB, EQ, NEQ, LT, LTE, GT, GTE, AND, NAND, OR,
                       NOR, XOR, XNOR, NOT )}
    = ir[20:16:-1] ;

Logic flagz [1] = flag[0];

// instruction fields
Logic f_loc [CODE_ADDR_SIZE] = ir[15:0:-1] ;
Logic f_add [8] = ir[7:0:-1];
Logic f_val [8] = ir[7:0:-1];
```

Dynamic section (Actions+Table)

```
Action incrpc   : pc := pc + 1 ;
Action wrram    : ram[f_add] := acc ;
Action rdrom    : ir := rom[pc] ;
Action setacc   : acc := f_val ;
Action calc     :  acc := case icode of {
    ADD : acc + acc2 ;
    SUB : acc - acc2 ;
    AND : acc & acc2 ;
    NAND : not(acc & acc2) ;
    OR : acc | acc2 ;
    NOR : not( acc | acc2) ;
    XOR : acc ^ acc2 ;
    XNOR : not( acc ^ acc2) ;
    NOT :  not acc ;
} ;
Action calc_flag     :  flag := case icode of {
    EQ : acc == acc2 ;
    NEQ : acc != acc2 ;
    LT : acc < acc2 ;
    LTE : acc <= acc2 ;
    GT : acc > acc2 ;
    GTE : acc >= acc2 ;
} ;
```

```
Table table(state,icode -> state ) {
    IDLE,  any, go      -> FETCH ;
    IDLE,  any, not go  -> IDLE ;
    FETCH, any      -> EXE, rdrom ;
    EXE, NOP        -> FETCH, incrpc ;
    EXE, LOAD       -> STALL_LOAD, ram.read(f_add) ;
    STALL_LOAD, any -> FETCH, acc := ram.dout, incrpc ;
    EXE, STORE      -> FETCH, wrram, incrpc ;
    EXE, SETV       -> FETCH, setacc, incrpc ;
    EXE, [ADD , SUB , AND , NAND , OR , NOR , XOR , XNOR]  -> STALL_ALU, ram.read(f_add) ;
    STALL_ALU, any  ->  EXE_ALU, acc2 := ram.dout ;
    EXE_ALU, any -> FETCH, calc , incrpc ;
    EXE, NOT     -> FETCH, calc , incrpc ;
    EXE, [EQ , NEQ , LT , LTE , GT , GTE]  -> STALL_FLAG, ram.read(f_add) ;
    STALL_FLAG , any -> EXE_ALU_FLAG , acc2 := ram.dout ;
    EXE_ALU_FLAG, any -> FETCH, calc_flag, incrpc ;
    EXE, JMP  -> FETCH, pc := f_loc ;
    EXE, JMPR -> FETCH, pc := pc+f_val ;
    EXE, JMPZ,   flagz  -> FETCH, pc := f_loc -> FETCH, incrpc ;
    //EXE, JMPZ,  not flagz  -> FETCH, incrpc ;
    EXE, JMPNZ, not flagz  -> FETCH, pc := f_loc -> FETCH, incrpc ;
    //EXE, JMPNZ,  flagz  -> FETCH, incrpc ;
    EXE, IJMPR -> FETCH, pc := pc + acc ;
    EXE, IJMP  -> FETCH, pc := acc ;
    EXE, HALT  -> DONE ;
    EXE, any   -> ERROR ;
    DONE, any  -> DONE ;
    ERROR, any -> ERROR
}
```

Conclusion

- «Beyond Verilog» abstraction is moving towards higher level software concepts, but hardware engineers are uncomfortable on that.
- An intermediate level of representation for digital systems is needed to allow designers to maintain strict control over hardware fundamentals.
- A possible approach is to capture the correct hardware semantics using a domain-specific language methodology .
- “Big opportunities for software engineers to help hardware designers in embracing a growing complexity” (Truong, Hanrahan)

C8B2

Closing

ldesantis@micron.com

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