Meta-HDL: A Multi-Stage Programming Language for Dynamically Reconfigurable Hardware

Adam Megacz
UC Berkeley Statistical Computing Facility
megacz@stat.berkeley.edu

Abstract

We present Meta-HDL, a two-stage functional language for writing software which dynamically synthesizes and communicates with structurally specialized circuits. The associated type system ensures that compiled programs cannot attempt to generate non-synthesizable logic. We implement the system by modifying the MetaOCaml bytecode compiler/interpreter to enforce our typing restrictions and emit synthesizable JHDL netlists at runtime. Finally, we examine two sample programs which demonstrate the wide degree to which the generated circuits can vary based on runtime inputs.

1 Overview

It is well documented that for a large variety of tasks, reconfigurable logic such as FPGAs can drastically outperform software running on fixed-purpose processors manufactured with the same technology [GNVV04]. Particular among these tasks is scenarios where part of the input to the computation is available ahead of time, or remains fixed during many iterations of the computation.

Despite these advantages, reconfigurable logic is only very rarely used in mainstream computing, even in scenarios where it would be especially superior to software.

1.1 Existing Work

The main impediment to widespread utilization of reconfigurable logic is the relative difficulty of writing programs for it. The current generation of reconfigurable logic tools generally fall into three categories:

- Tools which treat reconfigurable logic as a lower-density form of traditional hardware and re-use tools designed for that medium (VHDL [LMS86], Verilog [TM95]).
- Tools which attempt to translate code written in popular, imperative programming languages to circuitry [Ana] [Sod98]
- Tools which choose a software model that makes data dependencies clear and does not encourage the programmer to introduce artificial sequentialization [BCSS98] [SM01a] [LL95] [GL95] [LLC99].

The first class of tools includes those which work on conventional hardware design languages such as VHDL and Verilog. These tools give the designer exceptionally detailed control over the resulting circuitry, offering extremely space-efficient designs. However, developing a circuit with these tools is a much more labor intensive task than writing software, and the skills required are much less commonplace.

The second class of tools attempts to translate programs written in popular imperative languages into circuitry. These tools are quite easy to use since the programmer does not need to learn a new language, but the realization in hardware is often of extremely poor quality, since heap structures (values of unbounded size) are notoriously difficult to express spatially.

The third class of tools generally involves functional languages with restrictions to prevent unbounded values and unbounded recursion. These tools are also very easy to use and tend to lead to higher quality circuits since the programmer is aware of data dependencies and is not encouraged to introduce artificial serialization. Current work in this area has been focused mainly on the design of static logic – logic which is not specialized at runtime based on some portion of the inputs. Existing work on runtime circuit specialization has so far been limited to constant propagation [MS98] [SM01b]. We are not aware of any work which makes structural modifications at runtime beyond replacing gates with constant inputs.

1.2 Barriers to Runtime Circuit Generation

One major barrier to dynamic logic synthesis in this third class of tools is the lack of a staged type system. Without a staged type system and cross-stage persistence, the compiler cannot guarantee that recursive functions will not "leak" from the first stage (software) into synthesized logic, where they cannot be realized efficiently.

One possible solution is to impose the same heap and recursion restrictions on both stages (software as well

as hardware). However, some of the most interesting applications of runtime circuit synthesis involve the two stages cooperating, with each performing the task it is best suited for. Software code performs tasks which require general recursion and unbounded heaps, using these facilities to generate highly customized circuits which then in turn execute massively parallel computations very efficiently. We will present such an example later in this paper.

This model of cooperating stages was first introduced in the software world with GeHB [TEX03], a language in which a first stage program constructs a second stage program with bounded heap usage suitable for real-time applications.

2 Meta-HDL

Like MetaOCaml, Meta-HDL is an ML-derived language allows a software programmer to use *staging annotations* [TS97] to indicate that certain parts of a program belong to a different *stage*. These annotated blocks are first-class values which can be manipulated and composed as the program executes.

In MetaOCaml, the `!.` operator indicates that the program should compile a composed block into machine code and execute it. In Meta-HDL, this operator instead synthesizes circuitry from the composed block and downloads it to the FPGA. This operation returns a function which can be used like any other function; it transparently sends its arguments to the reconfigurable fabric for processing and returns values drawn from the FPGA's output lines.

As the code inside the staging annotations uses a subset of the full host language, there is no need for the programmer to learn a new language. Furthermore, passing data between software and hardware using *cross-stage persistence* is completely transparent. This makes it quite easy to test different positionings of the hardware/software boundary. Meta-HDL shares MetaOCaml's *environment classifier inference* [TN03] which ensures that the `!.` operator will never be applied to open code (expressions with unbound variables).

Validity of circuits is checked at compile time. Although the structure of circuits generated at runtime can vary greatly depending on the inputs, there is no need for runtime checks to ensure synthesizability.

2.1 Goals

Meta-HDL is not intended to be a comprehensive hardware design language, or even the beginnings of one. It deliberately sacrifices detailed control over synthesis in order to present the programmer with an environment in which the second-stage (hardware) language is a strict subset of the types and semantics of the familiar first-stage software language.

Currently, the barrier to adoption of reconfigurable logic in the mainstream computing world is due mainly to a scarcity of hardware design skills rather than to the limitations of available programmable logic devices. Meta-HDL attempts to trade off space-efficient logic mapping in favor of a programming model that is accessible and intuitive to the much larger community of software programmers.

The model of using cross-stage persistence for hardware-software communication was chosen in anticipation of recent work in FPGA interconnects for commodity hardware. Recent research projects such as Pilchard [LLC+01] and TKDM [PP03] attach reconfigurable logic directly to the host CPU's memory bus. This makes communication with the FPGA as efficient as function calls between different parts of a program. Since the costs of software-hardware communication in these models are identical to the cost of software-to-software function calls, it is reasonable to hide this distinction from the programmer without an unreasonable risk of introducing subtle performance bottlenecks.

2.2 Core Contribution

This paper builds on existing work in synthesizing hardware from functional languages [BCSS98] [SM01a] [LL95] [GL95] [LLC99] by extending those models, providing three core contributions:

- A two stage language which enables synthesis of hardware at runtime, including full *structural* specialization. Ground value types (integers, floats, tuples) are identical in both stages, enabling cross-stage persistence as an intuitive model for communication between hardware and software.
- A formal specification of the type system and a mapping of the language onto SAFL+. Together these two formalizations prove that typechecked programs cannot attempt to synthesize invalid hardware at runtime.
- An implementation of the language using the MetaOCaml compiler [CTHL03] which has been modified to emit JHDL [BH98] program fragments which are in turn simulated and synthesized.

3 Multi-stage Programming

Multi-stage languages [NN92, JGS93, Tah99] provide light-weight, high-level *annotations* that allow the programmer to break down computations into distinct *stages*. This facility supports a natural and algorithmic approach to program generation, where generation occurs in a first stage, and the synthesized program is executed during a second stage. The annotations are a small set of constructs for the construction, combination, and execution of delayed computations. Standard problems associated with program generation problems, such accidental variable capture and the representation of programs, are completely hidden from the programmer (c.f. [Tah99]).

The following simple program illustrates the three key constructs provided by the multi-stage language MetaOCaml [CTHL03]:

```
let rec power n x =
  if n=0 then <1>
         else <~x * ~(power (n-1) x)>
let power3 = <fun x -> ~(power 3 <x>)>
```

Ignoring the staging annotations (brackets <e> and escapes ~e, the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^3. Without staging, the last step simply returns a function that would invoke the `power` function every time it gets invoked with a value for x. In contrast, the staged version will allow us to build a function that computes the third power directly (that is, using only multiplication). To see how the staged annotations work, we can start from the last statement in the code above. Whereas a term `fun x ->` e `x` is a value, an annotated term <`fun x ->` ~(e <`x`>)> is not. Brackets mean that we want to construct a future stage computation, and escapes mean that we want to perform an immediate computation *while* building the bracketed computation. In a multi-stage language, these annotations are not hints, they are imperatives. Thus, the application e <`x`> must to be performed even though `x` is still an uninstantiated symbol. In the `power` example, `power 3` <`x`> is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the function `power`, the recursive application of `power` is escaped to ensure its immediate execution in the first stage. Evaluating the definition of `power3` produces

<`fun x -> x*x*x*1`>.

General-purpose multi-stage languages provide strong safety guarantees. For example, a program generator written in such a language is not only type-safe in the traditional sense, but the type system also guarantees that *any generated program will be type safe*. In this work, it is advocated as a practical, high-level approach for taking advantage of dynamically configurable hardware.

4 Formal Semantics of Meta-HDL

4.1 Syntax

The grammar for Meta-HDL is provided in Figure 1.

4.2 Type System

The typing judgements for Meta-HDL appear in Figure 2. Each type consists of a tuple (t, n) where t is the type of the term and n is the stage (0 or 1) in which the typing is valid.

The key points to notice in the type system are that recursive let-bindings (LETREC) are only valid in the first stage, and cross stage persistence (VARN) is only permitted for ground types.

As a derivitave of MetaOCaml, our implementation inherits environment classifier inference [TN03], which is required in order to ensure that the `run` operator is not applied to open code (expressions with unbound variables). The judgement for environment classifiers are not included in the type system shown here.

4.3 Soundness

The type system we present ensures that no term in the second stage will be bound to a recursive function, which the most crucial property for ensuring synthesizability. To complete the proof that the resulting circuitry is realizable, we provide a formal translation (Figure 4) from the grammar of valid second-level Meta-HDL terms into the grammar of SAFL+ [SM01a], a language for which it has been established that all programs can be synthesized.

In particular, our type system has ensured that the only operators which can persist into the second stage are primitive operators (those in the SAFL+ set a), so our type judgements ensure the validity of the last line of the translation ruleset. Additionally, by combination of the BRAC and ESC typing rules we know that $\sim e$ will evaluate to a valid e^1 term, so the translation of $\sim e$ is simply the translation of the result of evaluating e in the first stage.

$$
\begin{array}{rcll}
e & ::= & x & (4)\\
 & | & i & (5)\\
 & | & a(e_1, ..., e_k) & (6)\\
 & | & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & (7)\\
 & | & \text{let } x = e \text{ in } e_0 & (8)\\
 & & & (9)
\end{array}
$$

Figure 3: A closed subset of the grammar for SAFL+

$$
\begin{array}{rcll}
i & \rightarrow & i & (10)\\
x & \rightarrow & x & (11)\\
\text{true} & \rightarrow & 1 & (12)\\
\text{false} & \rightarrow & 0 & (13)\\
\text{let } x = e \text{ in } e & \rightarrow & \text{let } x = e \text{ in } e & (14)\\
\pi_i e & \rightarrow & a_\pi(e) & (15)\\
(e, e) & \rightarrow & a_{\text{pair}}(e, e) & (16)\\
\sim (e^0) & \rightarrow & e^{\hookrightarrow \star} & (17)\\
ee & \rightarrow & a_e(e) & (18)\\
 & & & (19)
\end{array}
$$

Figure 4: Translation from level-two Meta-HDL terms to SAFL+ terms

$$t ::= \text{int}|\text{bool}|\text{float}|t \to t|t \star t \quad (1)$$

$$e^1 ::= i|f|\ \text{true}\ |\ \text{false}\ |\texttt{+}|\texttt{-}|\texttt{*}|\texttt{+.}|\texttt{-.}|\texttt{*.}|x|\text{let } x = e \text{ in } e|ee|(e,e)|\pi_i e|\sim (e^0) \quad (2)$$

$$e^0 ::= i|f|\ \text{true}\ |\ \text{false}\ |\texttt{+}|\texttt{-}|\texttt{*}|\texttt{+.}|\texttt{-.}|\texttt{*.}|x|\text{let } x = e \text{ in } e|ee|(e,e)|\pi_i e|\lambda x.e|\text{let rec } x = e \text{ in } e|\langle e^1\rangle|\texttt{run}\ \ e \quad (3)$$

Figure 1: The Grammar for Meta-HDL

$$\frac{}{\Gamma \vdash i : (\ \text{int}\ , n)}(INT) \qquad \frac{}{\Gamma \vdash \{\texttt{*}|\texttt{+}|\texttt{-}\} : (\ \text{int}\ \to\ \text{int}, n)}(IPRIM)$$

$$\frac{}{\Gamma \vdash f : (\ \text{float}\ , n)}(FLOAT) \qquad \frac{}{\Gamma \vdash \{\texttt{*.}|\texttt{+.}|\texttt{-.}\} : (\ \text{float}\ \to\ \text{float}, n)}(FPRIM)$$

$$\frac{}{\Gamma \vdash \{\text{true}, \text{false}\} : (\ \text{bool}\ , n)}(BOOL) \qquad \frac{\Gamma \vdash e_0 : (\text{bool}, n) \qquad \Gamma \vdash e_1 : (t_1, n) \qquad \Gamma \vdash e_2 : (t_1, n)}{\Gamma \vdash (\ \text{if } e_0 \text{ then } e_1 \text{ else } e_2) : (t_1, n)}(IF)$$

$$\frac{\Gamma \vdash e : (t, 1)}{\Gamma \vdash \langle e\rangle : (\langle t\rangle, 0)}(BRAC) \qquad \frac{\Gamma \vdash e : (\langle t\rangle, 0)}{\Gamma \vdash \sim e : (t, 1)}(ESC) \qquad \frac{\Gamma \vdash e : (\langle t\rangle, 0)}{\Gamma \vdash (\texttt{run}\ \ e) : (t, 0)}(RUN)$$

$$\frac{\Gamma(x) = (t' ::= \{\text{int}|\text{bool}|\text{float}|t' \star t'\}, 0)}{\Gamma \vdash x : (t, n)}(VARN) \qquad \frac{\Gamma(x) = (t, n)}{\Gamma \vdash x : (t, n)}(VARL)$$

$$\frac{\Gamma \vdash e_0 : (\text{bool}, n) \qquad \Gamma \vdash e_1 : (t_1, n)}{\Gamma \vdash (e_0, e_1) : (t_0 \star t_1, n)}(PAIR) \qquad \frac{\Gamma \vdash e : (t_1 \star t_2, n)}{\Gamma \vdash \pi_i e : t_i}(PI)$$

$$\frac{\Gamma(x) = (t_0, n) \qquad \Gamma, x : (t_0, n) \vdash e : (t_1, n)}{\Gamma \vdash \lambda x.e : (t_0, 0) \to (t_1, 0)}(LAM) \qquad \frac{\Gamma \vdash e_0 : (t_1, n) \to (t_2, n) \qquad \Gamma \vdash e_1 : (t_1, n)}{\Gamma \vdash (e_0 e_1) : (t_2, n)}(APP)$$

$$\frac{\Gamma \vdash e_0 : (t_0, n) \qquad \Gamma; x : (t_0, n) \vdash e_1 : (t_1, n)}{\Gamma \vdash (\ \text{let } x = e_0 \text{ in } e_1) : (t_1, n)}(LET)$$

$$\frac{\Gamma(f) = (t_0, 0) \to t_1; x : (t_0, 0) \vdash e_0 : (t_1, 0) \qquad \Gamma, f : (t_1 \to t_2, 0) \vdash e_1 : (t_2, 0)}{\Gamma \vdash\ \text{let rec } f(x) = e_0 \text{ in } e_1 : (t_2, 0)}(LETREC)$$

Figure 2: The Meta-HDL Type System

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.Virtex.*;

public class MyCircuit extends Logic {
  public static CellInterface[]
      cell_interface = {
      in("a0_1_1011", 64),
      in("a1_2_1012", 64),
      out("out", 1024),
  };
   public MyCircuit(Node parent,
                    Wire a0_1_1011,
                    Wire a1_2_1012,
                    Wire out) {
     super(parent);
     connect("out", out);
     connect("a0_1_1011", a0_1_1011);
     connect("a1_2_1012", a1_2_1012);
     connect("a2_3_1013", a2_3_1013);
     Wire param_1857 = z_17_1027;
     Wire s_1600 = param_1857.range(63, 32);
     Wire r_1599 = param_1857.range(31, 0);
     Wire sym1 = wire(32);
     new FPMult(this, s_20_1600,
                constant(this, new BV(32,
                Float.floatToIntBits(1.226))),
                sym1, nc(1), 8, 0);
      // ....
```

Figure 5: Sample JHDL output

5 Implementation

We have implemented the Meta-HDL system by adding additional restrictions to MetaOCaml's type checking code to ensure typability under the judgements presented in Figure 2.

The internal implementation of the `!.` operator (also called `run`) was modified to spawn a JHDL process and feed it the appropriate code to construct the circuit. Figure 5 shows a sample fragment of this intermediate code.

```
(* good *)
let rec permute k =
  if k = 1 then ⟨ fun n → n ⟩
           else ⟨ fun n → n *
                  ~(permute (k-1)) (n-1)) ⟩
```

Figure 6: A naive staging of the `permute` function

```
(* better; splits the problem space in half on each recursion *)
let rec permute k =
  if k = 1 then
      ⟨ fun n → n ⟩
  else if k mod 2 = 0 then
      ⟨ fun n → ( ~(permute k/2) n) *
                ( ~(permute k/2) (n - ~(lift k/2))) ⟩
  else
      ⟨ fun n → n * ( ~(permute (k-1)) (n-1)) ⟩
```

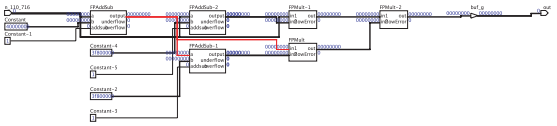

Figure 7: A better staging of `permute` with a depth of $\log(k)$ multipliers

5.1 Examples

5.1.1 Permutation Computation

Figures 6, 7, and 8 show three different stagings of the function `permute`, which computes $n \mathrm{P}\, k = \frac{n!}{(n-k)!}$. The functions generate a circuit specialized to a particular value of k.

The first circuit (Figure 6) demonstrates a naive staging fo the `permute` function. If executed in software on a serial processor, this staging would perform as well as the others. However, since each iteration of the recursion depends on all previous iterations, the critical path of the circuit contains $O(k)$ multiplications in sieres. This will yield a poor propagation delay.

The second circuit (Figure 7) shows an improved staging, where each recursion has an airty of 2, splitting the problem space in half. Although a programmer without hardware experience cannot be expected to understand the details of gate placement and propagation delay, it is quite easy to see that the two branches of the recursion do not depend on each other. Thus it is reasonable to expect a programmer with a minimal understanding of multithreaded or parallel programming to be able to pick out and eliminate data dependencies in a program, resulting in a circuit with a shorter critical path.

Although the second circuit has only $O(\log k)$ multiplications on the critical path, it unfortunately threads the decrement of the n argument serially through all of the computations. The final staging (Figure 8) eliminates this dependency by precomputing the constant values to subtract from n in each of the iteration. This allows all of the add/subtract operations to be performed in a single $O(1)$-time parallel operation, followed by $O(\log k)$ multiplications.

Mycroft and Sharp discuss source-to-source transfor-

```
(* best; lifts the computation of the subtractive
   amounts into the first stage *)
let rec permute k r =
      if k = 1          then ⟨ fun n → n − ∼(lift r) ⟩
  else if k mod 2 = 0 then
    ⟨ fun n →
      ( ∼(permute (k/2) (r+k/2)) n)
        *
        ( ∼(permute (k/2) r) n) ⟩
  else
        ⟨ fun n → n *
          ∼(permute (k−1) (r+1)) (n− ∼(lift r)) ⟩
```

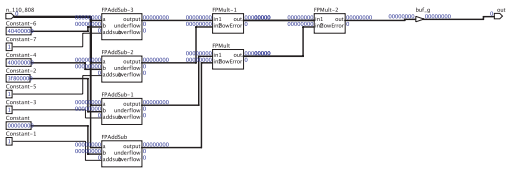

Figure 8: A third staging which eliminates nonessential data dependencies

mations as a way of exploring the design space for isolated circuits; Meta-HDL extends this technique to encompass the exploration of the space of possible hardware/software partitionings [1].

5.1.2 Variable-Radix Fast Fourier Transform

Figure 9 shows the Meta-HDL source code for a more complex example. This code constructs a Fast Fourier Transform which can be specialized to an arbitrary radix and arbitrary input window size at circuit synthesis time. Below the code is a circuit diagram for logic synthesized by running this code with `radix=2` on an input window of 16 complex numbers.

Removing the `memoize` and `memoized_continuation` functions would leave a perfectly normal single-stage OCaml program (athough it has been written in continuation passing style to make circuit-sharing easier). This program can then be staged by adding only these few lines of code to insert the staging annotations.

This program also performs *staged memoization* by let-binding the results of a computation and executing a continuation within the binding. This technique was pioneered by (*citation forthcoming*) at Rice University.

6 Future Directions

In its current form, Meta-HDL is only useful for combinatorial circuits with no feedback. We would like to extend this to accomodate tail recursion in the second stage by building on the techniques pioneered in SAFL+.

Subsequent work will focus on incrementally removing the restrictions on what kind of values can be present in the second (hardware) stage. Each increment will require both new techniques for realizing the functions in hardware as well as a refined type system to permit these new values while continuing to prohibit those which still cannot be realized. The relaxations we would like to attempt are:

- Permitting higher-order tail-call functions, but without the ability to form new closures in hardware. Initial investigations suggest that this could be done by joining the sub-circuits representing the individual functions with a bus in order to transfer control between arbitrary pairs of functions. However, this makes parallelization much more complex.
- Permitting non-tail recursion. So long as no heap structures (recursive values and values with unbounded size) can be created in the second stage, memory allocation is strictly LIFO, and can be modeled as a stack implemented as a massive shift register. Unlike heap allocation, this sort of memory management can be done quite efficiently in hardware.
- Bounded heap usage enforced by linear *capability types* in the same vein as LFPL. Existing work [TEX03] shows how to construct a two stage type system for such types.

On a more practical level, runtime placement and routing is a major barrier to Meta-HDL and all other runtime synthesis systems. Current work in the area of hardware-assisted fast routing [DHW02] [HWD03] looks promising. Since place and route algorithms are extremely complex, we intend to investigate whether or not using a language with formal semantics such as Meta-HDL could simplify the development of such a router, effectively using Meta-HDL to bootstrap its own synthesis toolchain.

7 Acknowledgements

Walid Taha provided the text for section 3 and proposed the idea of using an FFT as an example. He also came up with the name “Meta-HDL”. I am extremely grateful for his guidance and feedback on this paper, as well as his foundational work in multi-stage programming, as cited above.

I would also like to thank Rick Kawin, Deborah Nolan, and the Statistical Computing Facility for the use of their computing hardware for our simulations.

1. Mycroft and Sharp allude to this possibility in [MS01]; Meta-HDL adds cross-stage persistence and a formal type system for this purpose and validates the feasibility of the technique

```
let omega j n = let z = 2. *. pi *. ((float j) /. (float n)) in (cos z , sin z)

let rec upto = function 0 → [] | n → (n−1)::upto (n−1)

let rec memoize n f k =
        if n < 0
        then k f
        else ⟨ let z = ∼(f n) in ∼ (memoize (n−1) (fun i → if i=n then ⟨z⟩ else f i) k) ⟩

let rec fft radix length inputs k =

    let rec butterfly length inputs k slice =
        let rec continuation memoized_inputs =
          butterfly length
                    (fun i → if i⋆radix/length == slice
                             then (memoized_inputs (i mod length/radix))
                             else inputs i)
                    k (slice−1)
        and memoized_continuation inputs = memoize (length/radix−1) inputs continuation
        in if (slice < 0)
           then k inputs
           else if (length/radix < radix)
                then memoized_continuation inputs
                else butterfly (length/radix)
                               (fun i → inputs (i + length ⋆ slice / radix))
                               (fun inputs → memoized_continuation (merge (length/radix) inputs))
                               (radix−1)

    and merge length inputs i =
      fold (+...) (map
                    (fun slice →
                        (inputs (slice ⋆ (length/radix) + (i mod (length/radix))))
                      *...
                        ((omega (slice ⋆ (i mod (length/radix))) length)
                      *..
                         (omega (radix ⋆ slice ⋆ i / length) radix)))
                    (upto radix))

    in butterfly length inputs (fun inputs' → k (merge length inputs'))
```

Figure 9: Meta-HDL code implementing a Variable-Radix Fast Fourier Transform

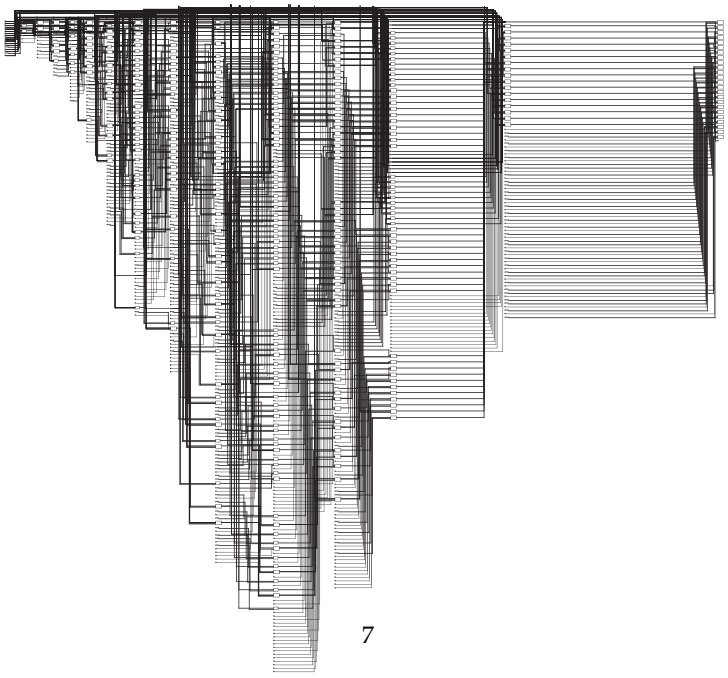

Figure 10: A circuit generated by the above code with `radix=2`

References

[Ana] C. Scott Ananian. Turning java into hardware: Caffinated compiler construction.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[BH98] Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[CTHL03] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[DHW02] Andre DeHon, Randy Huang, and John Wawrzynek. Hardware-assisted fast routing. 2002.

[GL95] Shaori Guo and Wayne Luk. Compiling Ruby into FPGAs. In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications*, pages 188–197. Springer-Verlag, Berlin, / 1995.

[GNVV04] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. 2004.

[HWD03] Randy Huang, John Wawrzynek, , and Andre DeHon. Stochastic, spatial routing for hypergraphs, trees, and meshes. 2003.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[LL95] Y. Li and M. Leeser. Hml: An innovative hardware description language and its translation to vhdl, 1995.

[LLC99] John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. In *Proceedings of the ACM SIGPLAN international conference on functional programming (ICFP '99), Paris, France, September 27–29, 1999*, volume 34(9), pages 60–69, New York, NY, USA, 1999. ACM Press.

[LLC+01] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee. Pilchard - a reconfigurable computing platform with memory slot interface, 2001.

[LMS86] R. Lipsett, E. Marschner, and M. Shaded. Vhdl - the language. ieee design and test of computers. 1986.

[MS98] N. McKay and S. Singh. Dynamic specialisation of XC6200 FPGAs by partial evaluation. *Lecture Notes in Computer Science*, 1482:298, 1998.

[MS01] Alan Mycroft and Richard Sharp. Hardware synthesis using SAFL and application to processor design. *Lecture Notes in Computer Science*, 2144:13+, 2001.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Number 34. Cambridge, 1992.

[PP03] Christian Plessl and Marco Platzner. Tkdm - a reconfigurable co-processor in a pc's memory slot. In *IEEE ICFPT*, 2003.

[SM01a] Richard Sharp and Alan Mycroft. A higher-level language for hardware synthesis. *Lecture Notes in Computer Science*, 2144:228, 2001.

[SM01b] Kong Woei Susanto and Thomas F. Melham. Formally analyzed dynamic synthesis of hardware. *The Journal of Supercomputing*, 19(1):7–22, 2001.

[Sod98] Donald Soderman. Implementing c designs in hardware. 1998.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from author's web-site.

[TEX03] Walid Taha, Stephan Ellner, and Hongwei Xi. Generating heap-bounded programs in a functional setting. In *EMSOFT*, 2003.

[TM95] Donald Thomas and Philip Moorby. 1995.

[TN03] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL*, 2003.

[TS97] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.