ABSTRACTING AN ERD-BODY

William Bricken
October 2002, revised October 2004

Abstraction is mandatory as designs grow large, since human designers find it exceedingly difficult to extract information from logic networks containing tens of thousands (or more) of gates. To accommodate this restriction, design itself has embodied modular principles. A primary example of this is bit-width: an 8-bit adder differs little from a 16 or 32-bit adder, except for replication of components. Thus modular abstraction already exists as a top-down design practice.

Top-down Abstraction

Nested modules provide an unlimited capacity for hierarchical abstraction and organization of a circuit design. By defining module templates in a top-down manner, a designer can rough-out the architecture of a design without having to specify the details. This is common practice during the specification phases of design. By assembling existing functional modules for specific behavioral components such as adders, comparators, buffers, counters, and the like, a designer can fill in certain aspects of an evolving design while leaving other aspects for later refinement. Thus nested modules provide valuable design tools that assist and facilitate designers during the conceptualization and specification phases of design, they provide organizational tools for the management of complexity of designs, and they enable to modular composition and reuse of design elements.

Figure I shows three stages in the top-down specification of a microprocessor. This figure serves as an example of the design process, and is not intended to illustrate a new design. At first, a designer may draw a very abstract functional block diagram, indicating the highest level units and the channels of communication flow. These functional blocks may then be articulated with some important concepts, such as bit-widths and types of data flow. As the design elaborates, more and more information may be filled in, some of it just placeholders, some it well understood, and perhaps some of it experimental, awaiting more information.

Similarly, Figure II shows that the ERD format can begin is a quite rough form, with details lacking and only functional modules present. Given sufficient guidance, the system can generate the abstract templates automatically and interactively, serving as a design drawing board, very much like software coding environments provide both templates and management of ideas. Figure III shows more detail of the ERD format being filled in, corresponding to the middle diagram in Figure I. Here emphasis is on the control cycles and steps of the FSM component, and the communication between the components. Functionality (here, CPU instructions) is expressed in module cells, while data paths are expressed in the input and output binding of the modules. Figure IV shows yet another level of top-down refinement, specifying more details about the data flow and instructions between the modules.

This example is not one of hardware design since it illustrates using the system capabilities for design prior to hardware specification. After the functionality has been defined at the conceptual and functional levels, a hardware engineer would then ask how to achieve this functionality in silicon. Figure V shows the ERD modular format for the cell-level hardware description of a functional microprocessor, for comparison. The microprocessor description is annotated by the number of each module type is incorporates when expanded. Thus the system can provide methods and support for the initial stages of the design cycle, as well as logic synthesis after the gate-level design is complete.

Bottom-Up Abstraction

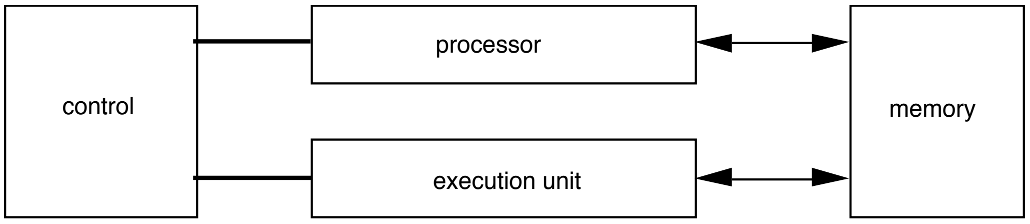
The methods of abstraction applied to parens forms generally apply to ERD-body format as well. The ERD format, however, allows more diverse types of symmetries to be abstracted than do parens forms. In particular, structural symmetries within parens forms must also be logical symmetries, since parens patterns are logic patterns. In the ERD-body two cell-forms can be symmetrical but not connected logically. Patterns across cells can identify functional equivalence that can be put into ERD module format, they can identify vectors of signals that travel together between logical structures, they can identify opportunities for structure sharing within the circuit, and they can identify glue logic subcircuits with no inherent structure. Each use is described herein.

Current EDA tools do not provide bottom-up abstraction, the ability to identify modular patterns in a circuit that have not necessarily been put there during design. The primary advantage of bottom-up abstraction is that designers can identify available trade-offs between circuit area and circuit delay. Another advantage of bottom-up abstraction is that modular components can be partitioned parametrically. Circuits can be sliced up in different ways, some of which may be more desirable for a particular design context. Not only does abstraction identify available modular components, it can move across different modular groupings, customizing the level and degree of abstraction.

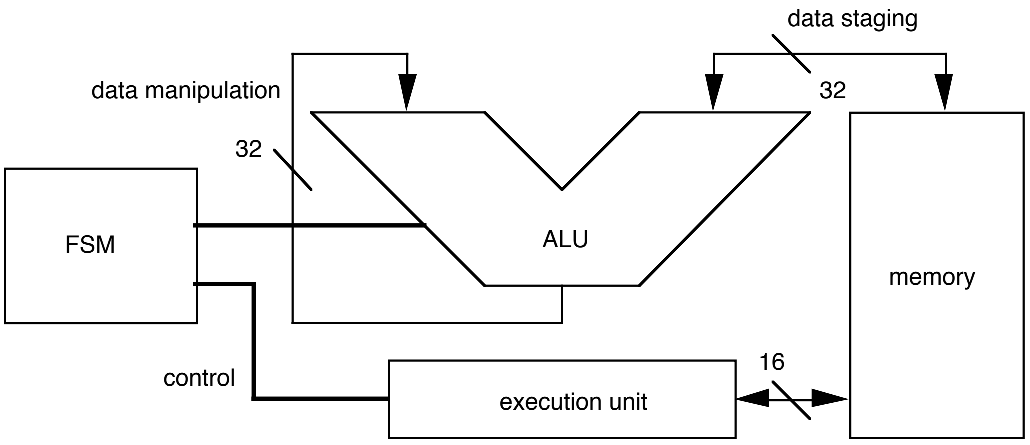
Although many tools can take a hierarchical design top-down and expand it into more fine-grain elements, the top-down design must first be defined and provided by a designer. The system can take netlist descriptions and construct bottom-up the abstract modules that form the circuit. This enhances ease of maintenance and of redesign. As well, poorly optimized circuits can be decomposed, re-optimized and reorganized automatically. Further, as illustrated by the following figures, the system can provide a diversity of decomposition and abstraction perspectives that permit a designer to explore specific aspects of the circuit structure such as logically necessary paths, wiring and routing demands, critical paths, and other design criteria. It should be noted that the small circuits used in these examples do not imply a limitation to the scalability of the techniques, methods, or processes incorporated in the system.

Figure I: An Example of a Top-down Design

Rough block diagram



More details 1



More details 2

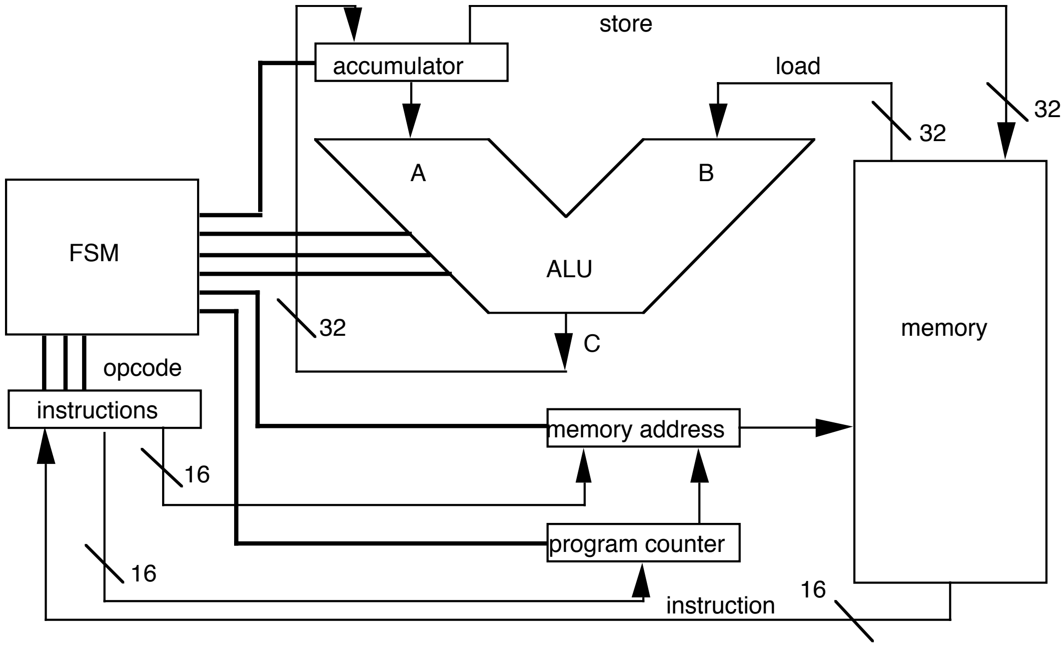


Figure II: An Example of a Module Template for the Top-down Design in Figure I, Rough Block Diagram

```
((design)
 ((main from-the-perspective-of-the-memory))
 ((proc-in unk)(exe-in1..32 unk)))
 ((proc-out 2)(exe-out 3))
 ((1-  control
        (())
        ((proc-out <?>)(exe-out <?>))   )
  (2-  processor
        ((control-in <?>)(memory-in <?>))
        ((memory-out <?>))   )
  (3-  execution-unit
        ((control-in <?>)(memory-in <?>))
        ((memory-out <?>))   )

((FSM-control)
 (())
 ((proc-out 0)(exe-out 1))
 ((0   <?>)
  (1   <?>  )  )  )

((ALU-processing)
 ((control-in unk)(memory-in <?>))
 ((memory-out 0))
 ((0 <?>)   )

((execution-unit)
 ((control-in unk)(memory-in <?>))
 ((memory-out 0))
 ((0 <?>)   )
```

Figure III: An Example of a Module Template for the Top-down Design in Figure I, More Details 1

```
((design)
 ((main from-the-perspective-of-the-memory))
 ((READ-address1..16 unk)(STORE-in1..32 unk)))
 ((INSTRUCTION-out1..16 READ)(LOAD-out1..32 READ))
 ((1FSM-  FSM-control
        ((RESETin unk)(new-instruction INSTRUCTION-out1..16))
        ((FETCH)(EXECUTE))   )
  (2ALU-  ALU-processing
        ((Ain  2ALU-cout1.32)(Bin  LOAD-out1)(OPin  INSTRUCTION-out1..16))
        ((Cout  2ALU-Ain1.32)(Aout STORE-in1..32))   )
  (3EXU-  execution-unit
        ((rin1..16  INSTRUCTION-out1..16))
        ((rout1..16  READ-address1..16)   )

((FSM-control)
 ((RESETin unk)(new-instruction unk))
 ((FETCH 2)(EXECUTE 7))
 ((0   RESETin)
  (1   RESET:  when 0 do 2 )
  (2   FETCH:  read-memory)
  (3   when READ do 4)
  (4   EXECUTE:  <op> -> ALU   )  )  )

((ALU-processing)
 ((Ain unk)(Bin unk)(OPin unk))
 ((Cout 0)(Aout ?))
 ((0 <A OPin B>)   )

((execution-unit)
 ((rin1..16 unk))
 ((rout1..16 0)(mout1..16 ?))
 ((0 rin1..16   )  )  )
```

Figure IV: An Example of a Module Template for the Top-down Design in Figure I, More Details 2

```
((design)
 ((main from-the-perspective-of-the-memory))
 ((READ-address1..16 unk)(STORE-in1..32 unk)))
 ((INSTRUCTION-out1..16 READ)(LOAD-out1..32 READ))
 ((1FSM-  FSM-control
        ((RESETin unk)(new-instruction INSTRUCTION-out1..16))
        ((FETCH)(EXECUTE))   )
  (2ALU-  ALU-processing
        ((Ain  6AC-rout1..32)(Bin  LOAD-out1)(OPin  INSTRUCTION-out1..16))
        ((Cout  6AC-rin1..32))   )
  (3IR-  16-bit-register
        ((rin1..16  INSTRUCTION-out1..16))
        ((rout1..16  4PCrin1..16)(rout1..16  5MR-rin1..16))   )
  (4PC-  16-bit-register
        ((rin1..16  3IR-rout1..16))
        ((rout1..16  5MR-rin1..16))   )
  (5MR-  16-bit-register
        ((rin1..16  3IR-rout1..16)(rin1..16 4PC-rout1..16)
        ((rout1..16  READ-address1..16))   )
  (6AC-  32-bit-register
        ((rin1..32  2ALU-Cout))
        ((rout1.32  2ALU-Ain)(rout1.32  STORE-in1..32))   )  ))

((FSM-control)
 ((RESETin unk)(new-instruction unk))
 ((FETCH 2)(FETCH 3)(FETCH 5)
  (EXECUTE 7)(EXECUTE 8)(EXECUTE 10)(EXECUTE 11)
  (EXECUTE 12)(EXECUTE 13)(EXECUTE 14))
 ((0   RESETin)
  (1   RESET:  when 0 do 2 then 3 then 4)
  (2   FETCH:  PC -> MR)
  (3   FETCH:  read-memory)
  (4   when READ do 5)
  (5   FETCH:  memory -> IR)
  (6   DECODE:  if 5 do 7 then 8)
  (7   EXECUTE:  IR -> MR)
  (8   EXECUTE:  read-memory)
  (9   when READ do 10 then 11 then 12 then 13 then 14 then 2)
  (10  EXECUTE:  LOAD -> Bin)
  (11  EXECUTE:  AC -> Ain)
  (12  EXECUTE:  <op> -> ALU)
  (13  EXECUTE:  Cout -> AC)
  (14  EXECUTE:  PC + 1)   )  )

((ALU-processing)
 ((Ain unk)(Bin unk)(OPin unk))
 ((Cout 0))
 ((0 <A OPin B>)   )

((16-bit-register)                          ((32-bit-register)
 ((rin1..16 unk))                                   ((rin1..32 unk))
 ((rout1..16 0))                                    ((rout1..32 0))
 ((0  rin1..16))   )                         ((0  rin1..32))   )
```

Figure V: Hierarchical Modularization of a CPU

```
                                                    instances    total

 ((main)                                                  --   1
  <input/output>
  (<registers>
   (23-..46-    aluslice  <bindings>)        -- 24
   (47-..62-    pcslice   <bindings>)        -- 16
   (63-..64-    decoder   <bindings>)        --  2
   (65-         encoder <bindings>)          --  1
   <main-functionality>))

  ((pcslice)                                              --  16
   <input/output>
   (<register>
    (21-..22-   m2        <bindings>)        --  2
    <pcslice-functionality>))

  ((aluslice)                                             --  24
   <input/output>
   (<register>
    (3-..13-    m2        <bindings>)        -- 11
    (14-..20-   regcell   <bindings>)        --  7
    <aluslice-functionality>)))

  ((regcell)                                              -- 168
   <input/output>
   (<register>
    (1-..2-     m2        <bindings>))       --  2

  ((m2)                                                   -- 662
   <input/output>
   (<m2-functionality>))

  ((encoder)                                              --   1
   <input/output>
   (<encoder-functionality>))

  ((decoder)                                              --   2
   <input/output>
   (<decoder-functionality>))
```