Computer Architecture: A Constructive Approach

Using Executable and Synthesizable Specifications

Arvind [1], *Rishiyur S. Nikhil* [2],
Joel S. Emer [3], *and Murali Vijayaraghavan* [1]

[1] MIT [2] Bluespec, Inc. [3] Intel and MIT

with contributions from

Prof. Li-Shiuan Peh, Abhinav Agarwal, Elliott Fleming,
Sang Woo Jun, Asif Khan, Myron King (MIT);
Prof. Derek Chiou (University of Texas, Austin);
and Prof. Jihong Kim (Seoul National University)

Revision: August 25, 2015

Acknowledgements

We would like to thank the staff and students of various recent offerings of this course at MIT, Seoul National University and Technion for their feedback and support.

Contents

Chapter 1

Introduction

This book is intended as an introductory course in Computer Architecture (or Computer Organization, or Computer Engineering) for undergraduate students who have had a basic introduction to circuits and digital electronics. This book employs a *constructive approach*, i.e., all concepts are explained with machine descriptions that transparently describe architecture and that can be synthesized to real hardware (for example, they can actually be run on FPGAs). Further, these descriptions will be very modular, enabling experimentation with alternatives.

Computer Architecture has traditionally been taught with schematic diagrams and explanatory text. Diagrams describe general structures, such as ALUs, pipelines, caches, virtual memory mechanisms, and interconnects, or specific examples of historic machines, such as the CDC 6600, Cray-1 (recent machines are usually proprietary, meaning the details of their structure may not be publicly available). These diagrams are accompanied by lectures and texts explaining principles of operation. Small quantitative exercises can be done by hand, such as measuring cache hit rates for various cache organizations on small synthetic instruction streams.

In 1991, with the publication of the classic *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson [6] (the Fith Edition was published in 2011), the pedagogy changed from such almost anecdotal descriptions to serious scientific, quantitative evaluation. They firmly established the idea that architectural proposals cannot be evaluated in the abstract, or on toy examples; they must be measured running real programs (applications, operating systems, databases, web servers, etc.) in order properly to evaluate the engineering trade-offs (cost, performance, power, and so on). Since it has typically not been feasible (due to lack of time, funds and skills) for students and researchers actually to build the hardware to test an architectural proposal, this evaluation has typically taken the route of simulation, i.e., writing a program (say in in C or C++) that simulates the architecture in question. Most of the papers in leading computer architecture conferences are supported by data gathered this way.

Unfortunately, there are several problems with simulators written in traditional programming languages like C and C++. First, it is very hard to write an accurate model of complex hardware. Computer system hardware is massively parallel (at the level of registers, pipeline stages, etc.); the paralleism is very fine-grain (at the level of individual bits and clock cycles); and the parallelism is quite heterogeneous (thousands of dissimilar activities). These features are not easy to express in traditional programming languages, and simulators that try to model these features end up as very complex programs. Further, in a simulator it is too easy, without realizing it, to code actions that are unrealistic or infeasible in real hardware, such as instantaneous, simultaneous access to a global piece of state from distributed parts of the architecture. Finally, these simulators are very far removed from representing any kind of formal specification of an architecture, which would be useful for both manual and automated reasoning about correctness, performance, power, equivalence, and so on. A formal semantics of the interfaces and behavior of architectural components would also benefit constructive experimentation, where one could more easily subtract, replace and add architectural components in a model in order to measure their effectiveness.

Of course, to give confidence in hardware feasibility, one could write a simulator in the synthesizable subset of a Hardware Description Language (HDL) such as Verilog, VHDL, or the RTL level of SystemC. Unfortunately, these are very low level languages compared to modern programming languages, requiring orders of magnitude more effort to develop, evolve and maintain simulators; they are certainly not good vehicles for experimentation. And, these languages also lack any useful notion of formal semantics.

In this book we pursue a recently available alternative. The BSV language is a high-level, fully synthesizable hardware description language with a strong formal semantic basis [1, 2, 9]. It is very suitable for describing architectures precisely and succinctly, and has all the conveniences of modern advanced programming languages such as expressive user-defined types, strong type checking, polymorphism, object orientation and even higher order functions during static elaboration.

BSV's behavioral model is based on Guarded Atomic Actions (or atomic transactional "rules"). Computer architectures are full of very subtle issues of concurrency and ordering, such as dealing correctly with data hazards or multiple branch predictors in a processor pipeline, or distributed cache coherence in scalable multiprocessors. BSV's formal behavioral model is one of the best vehicles with which to study and understand such topics (it seems also to be the computational model of choice in several other languages and tools for formal specification and analysis of complex hardware systems).

Modern hardware systems-on-a-chip (SoCs) have so much hardware on a single chip that it is useful to conceptualize them and analyze them as *distributed systems* rather than as globally synchronous systems (the traditional view), i.e., where architectural components are loosely coupled and communicate with messages, instead of attempting instantaneous access to global state. Again, BSV's formal semantics are well suited to this flavor of models.

The ability to describe hardware module interfaces formally in BSV facilitates creating reusable architectural components that enables quick experimentation with alternatives structures, reinforcing the "constructive approach" mentioned in this book's title.

Architectural models written in BSV are fully executable. They can be simulated in the Bluesim$^{\text{TM}}$ simulator; they can be synthesized to Verilog and simulated on a Verilog simulator; and they can be further synthesized to run on FPGAs, as illustrated in Fig. 1.1. This last capability is not only excellent for validating hardware feasibility of the models, but it also enables running *much bigger programs* on the models, since FPGA-based execution can be 3 to 4 *orders of magnitude* faster than simulation. Students are also very excited to see their designs actually running on real FPGA hardware.

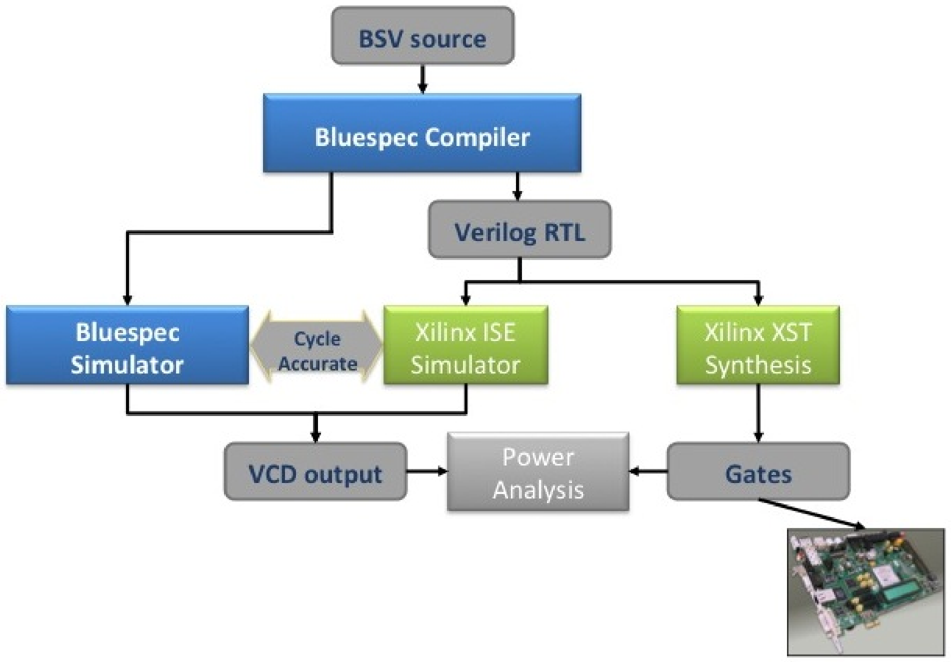


Figure 1.1: Tool flow for executing the models in this book

This book uses the SMIPS Instruction Set Architecture for all its examples, but none of the architectural concepts discussed here are specific to SMIPS. The choice of SMIPS is one of expedience—it is a simple, open ISA, for which a C compiler is available (without a compiler, it would be quite laborious to construct any serious programs to run on our implementations). And, in any case, no matter what ISA you are implementing, most of the architectural concepts are identical; only the front-end instruction-stream parsing and decode stages are likely to be different. We welcome others to adapt the examples provided here to other ISAs (preferably in open source form, available to the community).

In this book, intended as a first course in Computer Architecture for undergraduates, we will go through a series of traditional topics: combinational circuits, pipelines, unpipelined processors, processor pipelines of increasing depth, control speculation (branch prediction), data hazards, exceptions, caches, virtual memory, and so on. We will be writing BSV code for every topic explored. At every stage the student is encouraged to run the models at least in simulation, but preferably also on FPGAs. By the end of the course students will design six or more different computers of increasing complexity and performance, and they will quantitatively evaluate the performance of their C programs compiled and running on these machines. Codes will be written in a modular way so students can experiment by easily removing, replacing or adding components or features. Along the way, we will delve deeply into complex issues of concurrency and ordering so that the student has a more rigorous framework in which to think about these questions and create new solutions.

Chapter 2

Combinational circuits

Combinational circuits are just acyclic interconnections of gates such as AND, OR, NOT, XOR, NAND, and NOR. In this chapter, we will describe the design of some Arithmetic-Logic Units (ALUs) built out of pure combinational circuits. These are the core "processing" circuits in a processor. Along the way, we will also introduce some BSV notation, types and type checking.

2.1 A simple "ripple-carry" adder

Our first goal is to build an adder that takes two w-bit input integers and produces a w-bit output integer plus a "carry" bit (typically, $w = 32$ or 64 in modern processors). First we need a building block called a "full adder" whose inputs are three bits—the j^{th} bits of the two inputs and the "carry" bit from the addition of the lower-order bits (up to the $j - 1^{th}$ bit). The full adder has two output bits—the j^{th} bit of the result and the carry bit up to this point. Its functionality is specified by a classical "truth table":

Inputs			Outputs	
a	b	c_in	c_out	s
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

We can implement this in a circuit, expressed in BSV (almost correct syntax) as follows:

Full Adder (not quite proper BSV code)
```
function fa (a, b, c_in);
   s = (a ^ b) ^ c_in;
   c_out = (a & b) | (c_in & (a ^ b));
   return {c_out, s};
endfunction
```

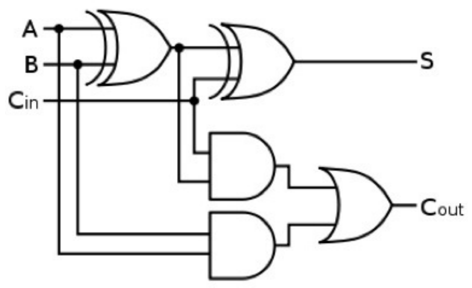


Figure 2.1: Full adder circuit

In BSV, as in C, the `&`, `|` and `^`, symbols stand for the AND, OR, and XOR (exclusive OR) functions on bits. The circuit it describes is shown in Fig. 2.1, which is in some sense just a pictorial depiction of the code.

A note about common sub-expressions: the code has two instances of the expression `(a ^ b)` (on lines 2 and 3), but the circuit shows just one such gate (top-left of the circuit), whose output is fed to two places. Conversely, the circuit could have been drawn with two instances of the gate, or the expression-sharing could have been suggested explicitly in the BSV code:

Common sub-expressions

```
   ...
   tmp = (a ^ b);
   s = tmp ^ c_in;
   c_out = (a & b) | (c_in & tmp);
   ...
```

From a functional point of view, these differences are irrelevant, since they all implement the same truth table. From an implementation point of view it is quite relevant, since a shared gate occupies less silicon but drives a heavier load. However, we do not worry about this issue at all when writing BSV code because the Bluespec compiler *bsc* performs extensive and powerful identification and unification of common-sub-expressions, no matter how you write it. Such sharing is almost always good for implementation. Very occasionally, considerations of high fan-out and long wire-lengths may dictate otherwise; BSV has ways of specifying this, if necessary. **(end of note)**

The code above is not quite proper BSV, because we have not yet declared the types of the arguments, results and intermediate variables. Here is a proper BSV version of the code:

Full Adder (with types)

```
function Bit#(2) fa (Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   Bit#(1) s = (a ^ b) ^ c_in;
   Bit#(1) c_out = (a & b) | (c_in & (a ^ b));
   return {c_out,s};
endfunction
```

BSV is a strongly typed language, following modern practice for robust, scalable programming. All expressions (including variables, functions, modules and interfaces) have unique types, and the compiler does extensive type checking to ensure that all operations have meaningful types. In the above code, we've declared the types of all arguments and the two intermediate variables as `Bit#(1)`, i.e., a 1-bit value. The result type of the function has been declared `Bit#(2)`, a 2-bit value. The expression `{c_out,s}` represents a concatenation of bits. The type checker ensures that all operations are meaningful, i.e., that the operand and result types are proper for the operator (for example, it would be meaningless to apply the square root function to an Ethernet header packet!).

BSV also has extensive *type deduction* or *type inference*, by which it can fill in types omitted by the user. Thus, the above code could also have been written as follows:

Full Adder (with some omitted types)

```
function Bit#(2) fa (Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   let s = (a ^ b) ^ c_in;
   let c_out = (a & b) | (c_in & (a ^ b));
   return {c_out,s};
endfunction
```

The keyword `let` indicates that we would like the compiler to work out the types for us. For example, knowing the types of `a`, `b` and `c_in`, and knowing that the `^` operator takes two w-bit arguments to return a w-bit result, the compiler can deduce that `s` has type `Bit#(1)`. Similarly, the compiler can also deduce that `c_out` has type `Bit#(1)`.

Type-checking in BSV is much stronger than in languages like C and C++. For example, one can write an assignment statement in C or C++ that adds a `char`, a `short`, a `long` and a `long long` and assign the result to a `short` variable. During the additions, the values are silently "promoted" to longer values and, during the assignment, the longer value is silently truncated to fit in the shorter container. The promotion may be done using zero-extension or sign-extension, depending on the types of the operands. This kind of silent type "casting" (with no visible type checking error) is the source of many subtle bugs; even C/C++ experts are often surprised by it. In hardware design, these errors are magnified by the fact that, unlike C/C++, we are not just working with a few fixed sizes of values (1, 2, 4 and 8 bytes) but with arbitrary bit widths in all kinds of combinations. Further, to minimize hardware these bit sizes are often chosen to be just large enough for the job, and so the chances of silent overlow and truncation are greater. In BSV, type casting is never silent, it must be explicitly stated by the user.

2.1.1 A 2-bit Ripple-Carry Adder

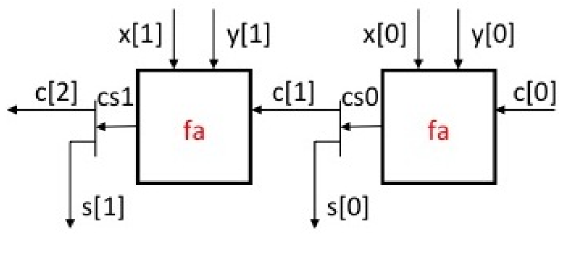


Figure 2.2: 2-bit Ripple Carry Adder circuit

We now use our Full Adder as a black box for our next stepping stone towards a w-bit ripple-carry adder: a 2-bit ripple-carry adder. The circuit is shown in Fig. 2.2, and it is described by the following BSV code.

2-bit Ripple Carry Adder
```
function Bit#(3) add(Bit#(2) x, Bit#(2) y, Bit#(1) c0);
   Bit#(2) s;
   Bit#(3) c = {?,c0};

   let cs0 = fa(x[0], y[0], c[0]);
   c[1] = cs0[1];  s[0] = cs0[0];

   let cs1 = fa(x[1], y[1], c[1]);
   c[2] = cs1[1];  s[1] = cs1[0];

   return {c[2],s};
endfunction
```

Here, `x` and `y` represent the two 2-bit inputs, and `c0` the input carry bit. The notation `x[`j`]` represents a selection of the j^{th} bit of `x`, with the common convention that $j = 0$ is the least significant bit. Fig. 2.3 shows the circuit for bit-selection where the selection index is

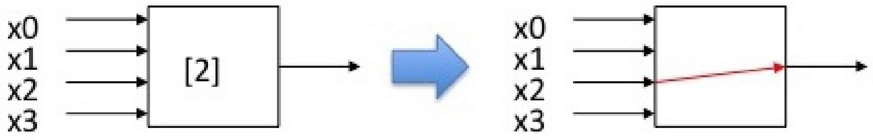


Figure 2.3: Bit-select circuit, static index

a static (compile-time) constant. The "circuit" is trivial: we just connect the specified wire. Fig. 2.3 shows the circuit for bit-selection where the selection index is a dynamic value.

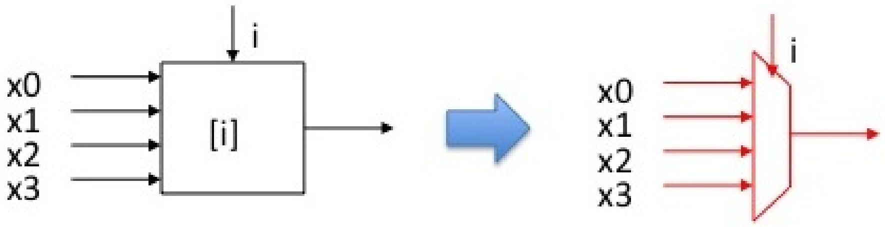


Figure 2.4: Bit-select circuit, dynamic index

Here, we need a "multiplexer", which we will discuss in Sec. 2.4.1. In the ripple-carry adders we discuss here, we only use bit-selection with static indices.

Returning to our 2-bit adder, we declare a 2-bit value `s` and a 3-bit value `c` to hold some intermediate values. The latter is initialized to the value of the expression `{?,c0}`. This is a bit-concatenation of two values, one that is left unspecified, and `c0`. The compiler can deduce that the unspecified value must be of type `Bit#(2)`, since the whole expression must have type `Bit#(3)` and `c0` has type `Bit#(1)`.

In BSV, the expression `?` represents an unspecified or "don't care" value, leaving it up to the compiler to choose. Here we leave it unspecified since we're going to set the values in the subsequent code. Similarly, the initial value of `s` is also unspecified by not even providing an initializer.

A full adder instance is applied to the two lower bits of `x` and `y` and the input carry bit producing 2 bits, `cs0`. We assign its bits into `c[1]` and `s[0]`. A second full adder instance is applied to the two upper bits of `x` and `y` and the carry bit of `cs0` to produce 2 bits, `cs1`. Again, we capture those bits in `c[2]` and `s[1]`. Finally, we construct the 3-bit result from `c[2]` and `s`. The type checker will be happy with this bit-concatenation since the widths add up correctly.

The types `Bit#(1)`, `Bit#(2)`, `Bit#(3)` etc. are instances of a more general type `Bit#(n)`, where `n` is a "type variable". Such types are variously called *parameterized types*, *polymorphic types*, or *generic types*, because they represent a class of types, corresponding to each possible concrete instantiation of the type variable.

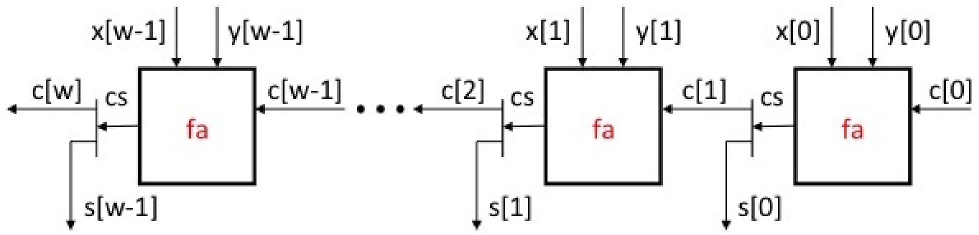


Figure 2.5: w-bit Ripple Carry Adder circuit

We are now ready to generalize our 2-bit ripple-carry adder to a w-bit ripple-carry adder. The circuit is shown in Fig. 2.5, and the BSV code is shown below.

w-bit Ripple Carry Adder (almost correct)

```
function Bit#(w+1) addN (Bit#(w) x, Bit#(w) y, Bit#(1) c0);
   Bit#(w) s;
   Bit#(w+1) c = {?, c0};
   for(Integer i=0; i<w; i=i+1) begin
      let cs = fa (x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
   return {c[w],s};
endfunction
```

The previous 2-bit adder can be seen as an explicit unrolling of this loop with $w = 2$. Now this generic w-bit adder can be instantiated as adders of specific sizes. For example:

w-bit Ripple Carry Adder ()

```
function Bit#(33) add32 (Bit#(32) x, Bit#(32) y, Bit#(1) c0)
    = addN (x,y,c0);

function Bit#(4) add3 (Bit#(3) x, Bit#(3) y, Bit#(1) c0)
    = addN (x,y,c0);
```

This defines add32 and add3 on 32-bit and 3-bit values, respectively. In each case the specific function just calls addN, but type deduction will fix the value of w for that instance, and type checking will verify that the output width is 1 more than the input width.

We will now correct a small notational issue with the addN function given above, having to do with numeric type notations. The BSV compiler, like most compilers, is composed of several distinct phases such as parsing, type checking, static elaboration, analysis and optimization, and code generation. Type checking must analyze, verify and deduce numerical relationships, such as the bit widths in the example above. However, we cannot have arbitrary arithmetic in this activity since we want type-checking to be feasible and efficient (arbitrary arithmetic would make it undecidable). Thus, the numeric calculations performed by the type checker are in a separate, limited universe, and should not be confused with ordinary arithmetic on values.

In the type expressions `Bit#(1)`, `Bit#(3)`, `Bit#(33)`, the literals `1`, `3`, `33`, are in the type universe, not the ordinary value universe, even though we use the same syntax as ordinary numeric values. The context will always make this distinction clear—numeric literal types only occur in type expressions, and numeric literal values only occur in ordinary value expressions. Similarly, it should be clear that a type variable like `w` in a type expression `Bit#(w)` can only stand for a numeric type, and never a numeric value.

Since type checking (including type deduction) occurs before anything else, type values are known to the compiler before it analyses any piece of code. Thus, it is possible to take numeric type values from the types universe into the ordinary value universe, and use them there (but not vice versa). The bridge is a built-in pseudo-function called `valueOf(n)`. Here, `n` is a numeric type expression, and the value of the function is an ordinary `Integer` *value* equivalent to the number represented by that type.

Numeric type expressions can also be manipulated by numeric type operators like `TAdd#(t1,t2)` and `TMul#(t1,t2)`, corresponding to addition and subtraction, respectively (other available operators include min, max, exponentiation, base-2 log, and so on).

With this in mind, we can fix up our w-bit ripple carry adder code:

`w-bit Ripple Carry Adder (corrected)`

```
1  function Bit#(TAdd#(w,1)) addN (Bit#(w) x, Bit#(w) y, Bit#(1) c0);
2     Bit#(w) s;
3     Bit#(TAdd#(w,1)) c = {?, c0};
4     for(Integer i=0; i< valueOf(w); i=i+1) begin
5        let cs = fa (x[i],y[i],c[i]);
6        c[i+1] = cs[1]; s[i] = cs[0];
7     end
8     return {c[w],s};
9  endfunction
```

The only differences from the earlier, almost correct version is that we have used `TAdd#(w,1)` instead of `w+1` in lines 1 and 3 and we have inserted `valueOf()` in line 4.

2.2 Static Elaboration and Static Values

The w-bit ripple carry adder in the previous section illustrated how we can use syntactic for-loops to represent repetitive circuit structures. Conceptually, the compiler "unfolds" such loops into an acyclic graph representing a (possibly large) combinational circuit. This process in the compiler is called *static elaboration*. In fact, in BSV you can even write recursive functions that will be statically unfolded to represent, for example, a tree-shaped circuit.

Of course, this unfolding needs to terminate statically (during compilation), i.e., the conditions that control the unfolding cannot depend on a dynamic, or run-time value (a value that is only known when the circuit itself runs in hardware, or is simulated in a simulator). For example, if we wrote a for-loop whose termination index was such a run-time value, or a recursive function whose base case test depended on such a run-time value, such a loop/function could not be unfolded statically. Thus, static elaboration of loops and recursion must only depend on static values (values known during compilation).

This distinction of static elaboration vs. dynamic execution is not something that software programmers typically think about, but it is an important topic in hardware design. In software, a recursive function is typically implemented by "unfolding" it into a stack of frames, and this stack grows and shrinks dynamically; in addition, data is communicated between frames, and computation is done on this data. In a corresponding hardware implementation, on the other hand, the function may be statically unfolded by the compiler into a tree of modules (corresponding to pre-elaborating the software stack of frames), and only the data communication and computation happens dynamically. Similarly, the for-loop in the w-bit adder, if written in C, would typically actually execute as a sequential loop, dynamically, whereas what we have seen is that our loop is statically expanded into repetitive hardware structures, and only the bits flow through it dynamically.

Of course, it is equally possible in hardware as well to implement recursive structures and loops dynamically (the former by pushing and popping frames or contexts in memory, the latter with FSMs), mimicing exactly what software does. In fact, later we shall see the BSV "FSM" sub-language where we can express dynamic sequential loops with dynamic bounds. But this discussion is intended to highlight the fact that hardware designers usually think much more carefully about what structures should/will be statically elaborated vs. what remains to execute dynamically.

To emphasize this point a little further, let us take a slight variation of our w-bit adder, in which we do *not* declare `c` as a `Bit#(TAdd#(w,1))` bit vector:

w-bit Ripple Carry Adder (variation)

```
function Bit#(w+1) addN (Bit#(w) x, Bit#(w) y, Bit#(1) c);
   Bit#(w) s;
   for(Integer i=0; i<w; i=i+1) begin
      let cs = fa (x[i],y[i],c);
      c = cs[1]; s[i] = cs[0];
   end
   return {c,s};
endfunction
```

Note that `c` is declared as an input parameter; it is "repeatedly updated" in the loop, and it is returned in the final result. The traditional software view of this is that `c` refers to a location in memory which is repeatedly updated as the loop is traversed sequentially, and whatever value finally remains in that location is returned in the result.

In BSV, `c` is just a name for value (there is no memory involved here, let alone any concept of `c` being a name for a location in memory that can be updated). The loop is statically elaborated, and the "update" of `c` is just a notational device to say that this is what `c` now means for the rest of the elaboration. In fact this code, and the previous version where `c` was declared as `Bit#(TAdd#(w,1))` are both statically elaborated into identical hardware.

Over time, it becomes second nature to the BSV programmer to think of variables in this way, i.e., not the traditional software view as an assignable location, but the purer view of being simply a name for a value during static elaboration (ultimately, a name for a set of wires).[1]

2.3 Integer types, conversion, extension and truncation

One particular value type in BSV, `Integer`, is only available as a static type for static variables. Semantically, these are true mathematical unbounded integers, not limited to any arbitrary bit width like 32, 64, or 64K (of course, they're ultimately limited by the memory of the system your compiler runs on). However, for dynamic integer values in hardware we typically limit them to a certain bit width, such as:

Various BSV fixed-width signed and unsigned integer types

```
int, Int #(32)    \\ 32-bit signed integers
Int #(23)         \\ 23-bit signed integers
Int #(w)          \\ signed integer of polymorphic width w
UInt #(48)        \\ 48-bit unsigned integers
...
```

One frequently sees `Integer` used in statically elaborated loops; the use of this type is a further reminder that this is a statically elaborated loop.

In keeping with BSV's philosophy of strong type checking, the compiler never performs automatic (silent) conversion between these various integer types; the user must express a desired conversion explicitly. To convert from `Integer` to a fixed-width integer type, one applies the `fromInteger` function. Examples:

fromInteger

```
for(Integer j=0; i< 100; j=j+1) begin
   Int #(32) x = fromInteger (j/4);
   UInt #(17) y = fromInteger (valueOf (w) + j);
   ...
end
```

[1]For compiler afficionados: this is a pure functional view, or Static Single Assignement (SSA) view of variables.

For truncation of one fixed-size integer type to a shorter one, use the `truncate` function. For extension to a longer type, use `zeroExtend`, `signExtend` and `extend`. The latter function will zero-extend for unsigned types and sign-extend for signed types. Examples:

extend and truncate

```
Int #(32) x = ...;
Int #(64) y = signExtend (x);
x = truncate (y);
y = extend (x+2);    // will sign extend
```

Finally, wherever you really need to, you can always convert one type to another. For example you might declare a memory address to be of type `UInt#(32)`, but in some cache you may want to treat bits [8:6] as a signed integer bank address. You'd use notation like the following:

pack and unpack

```
typedef UInt #(32) Addr;
typedef Int #(3)   BankAddr;

Addr     a  = ...
BankAddr ba = unpack (pack (a)[8:6]);
```

The first two lines define some type synonyms, i.e., `Addr` and `BankAddr` can now be regarded as more readable synonyms for `UInt#(32)` and `Int#(3)`. The `pack` function converts the `Int #(32)` value `a` into a `Bit#(32)` value; we then select bits [8:6] of this, yielding a `Bit#(3)` value; the `unpack` function then converts this into an `Int#(3)` value.

This last discussion is not a compromise on strong static type checking. Type conversions will always be necessary because one always plays application-specific representational tricks (how abtract values and structures are coded in bits) that the compiler cannot possible know about, particularly in hardware designs. However, by making type conversions explicit in the source code, we eliminate most of the accidental and obscure errors that creep in either due to weak type checking or due to silent implicit conversions.

2.4 Arithmetic-Logic Units (ALUs)

At the heart of any processor is an ALU (perhaps more than one) that performs all the additions, subtractions, multiplications, ANDs, ORs, comparisons, shifts, and so on, the basic operations on which all computations are built. In the previous sections we had a glimpse of building one such operation—an adder. In this section we look at a few more operations, which we then combine into an ALU.

Fig. 2.6 shows the symbol commonly used to represent ALUs in circuit schematics. It has data inputs A and B, and an Op input by which we select the function performed by the ALU. It has a Result data output, and perhaps other outputs like Comp? for the outputs of comparisons, for carry and overflow flags, and so on.

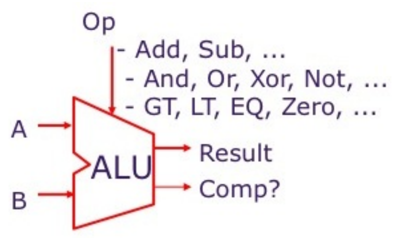


Figure 2.6: ALU schematic symbol

2.4.1 Shift operations

We will build up in stages to a general circuit for shifting a w-bit value right by n places.

Logical Shift Right by a Fixed Amount

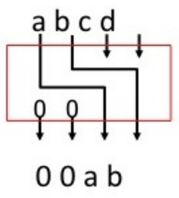


Figure 2.7: Logical shift right by 2

Fig. 2.7 shows a circuit for shifting a 4-bit value right by 2 (i.e., towards the least-significant bit), filling in zeroes in the just-vacated most-significant bits (in general the input and output could be w bits). As you can see, there's really no logic to it (`:-)`), it's just a wiring diagram. BSV has a built-in operator for this, inherited from Verilog and SystemVerilog:

Shift operators
```
   abcd >> 2    // right shift by 2
   abcd << 2    // left shift by 2
```

Even if it were not built-in, it is very easy to write a BSV function for this:

Shift function
```
function Bit #(w) logical_shift_right (Integer n, Bit #(w) arg);
   Bit #(w) result = 0;
   for (Integer j=0; j<(valueOf(w)-n); j=j+1)
      result [j] = arg [j+n];
   return result;
endfunction
```

This is statically elaborated to produce exactly the same circuit.

Other kinds of shifts—arithmetic shifts, rotations, etc.—are similar.

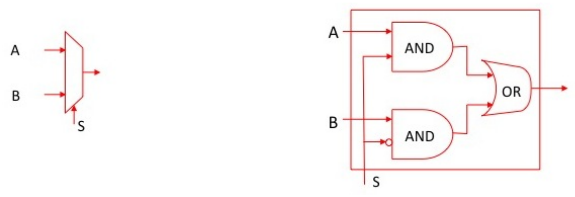


Figure 2.8: 2-way multiplexer

Multiplexers

A 2-way multiplexer ("mux") is a circuit that takes two inputs of type t and forwards one of them into the output of type t. The choice is made with a boolean control input. The left side of Fig. 2.8 shows an abstract symbol for a mux: one of the two inputs A or B is forwarded to the output depending on the value of the control input S. The right side of the figure shows an implementation in terms of AND and OR gates, for a 1-bit wide, 2-way mux (a w-bit wide mux would just replicate this w times, and tie all the control inputs together).

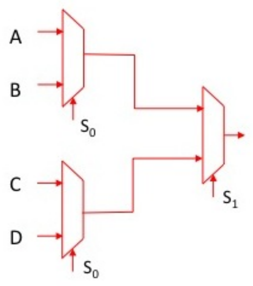


Figure 2.9: 4-way multiplexer

Larger muxes that select 1 out of n inputs ($n > 2$) can in principle be created merely by cascading 2-way muxes. For example, Fig. 2.9 shows a 4-way mux created using 2-way muxes. Of course, the previous boolean control line now generalizes to `Bit#(ln)` where `ln` is the base-2 logarithm of n (rounded up to the next integer), i.e., we need a `Bit#(ln)` control input to identify one of the n inputs to forward.

In BSV code we don't directly talk about muxes, but use classical programming language conditional expressions, if-then-else statements, or case statements:

Multiplexers
```
result = ( s ? a : b );

if (s)
   result = a;
else
   result = b;

case (s2)
   0: result = a;
   1: result = b;
```

```
        2: result = c;
        3: result = d;
    endcase
```

Of course, unlike conditional expressions, if-then-else and case statements can be more complex because you can have multiple statements in each of the arms. The differences in style are mostly a matter of readability and personal preference, since the compiler will produce the same hardware circuits anyway. Both *bsc* and downstream tools perform a lot of optimization on muxes, so there is generally no hardware advantage to writing it one way or another.

Notice that we said that the inputs and output of the mux are of some type t; we did not say `Bit#(w)`. Indeed, `Bit#(w)` is just a special case. In general, it is good style to preserve abstract types (Address, EthernetPacket, CRC, IPAddress, ...) rather than descending into bits all the time, in order to preserve readability, and the robustness that comes with strong type checking.

Be careful about unnecessary conditionals in your code (resulting in muxes). Multi-way muxes on wide datapaths can be very expensive in gates and cicuit latency, especially when implemented on FPGAs.

Logical Shift Right by n, a Dynamic Amount

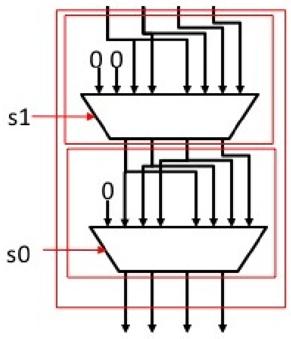


Figure 2.10: Shift by s = 0,1,2,3

We can now see how to implement a circuit that shifts a w-bit input value by s, where s is a dynamic value. The general scheme is illustrated in Fig. 2.10. Suppose the shift-amount is `s`, of type `Bit#(n)`. Then, we cascade n stages as follows:

- Stage 0: if `s[0]` is 0, pass through, else shift by 1 (2^0)
- Stage 1: if `s[1]` is 0, pass through, else shift by 2 (2^1)
- Stage 2: if `s[2]` is 0, pass through, else shift by 4 (2^2)
- ...
- Stage j: if `s[j]` is 0, pass through, else shift by 2^j
- ...

You can see that the total amount shifted is exactly what is specified by the value of `s`! Each stage is a simple application of constant shifting and multiplexing controlled by one bit of `s`. And the sequence of stages can be written as a for-loop.

You will be writing this BSV program as an exercise in the laboratory.

2.4.2 Enumerated types for expressing ALU opcodes

An ALU is a multi-purpose box. It typically has two input ports and an output port, and an additional control input port, the *opcode*, which specifies what function we want the ALU to perform on the inputs: addition, subtraction, multiplication, shifting, testing for zero, ... and so on. Of course the control input is represented in hardware as some number of bits, but following our preference to use modern programming language practice for better readability and robustness due to type checking, we define these as *enumerated types* with symbolic names.

Let's start with a simple, non-ALU example of enumerated types. Suppose we had a variable to identify colors on a screen—it takes one of three values: Red, Blue and Green. Of course, in hardware these will need some encoding, perhaps 00, 01 and 10, respectively. But in BSV we typically just say:

Enumerated types
```
typedef enum {Red, Blue, Green} Color
        deriving (Bits, Eq, FShow);
```

Line 1 is an enumerated type just like in C or C++, defining a new type called `Color` which has three constants called `Red`, `Blue` and `Green`. Line 2 is a BSV incantation that tells the compiler to define certain functions automatically on the type `Color`. The `Bits` part tells the compiler to define canonical bit representations for these constants. In fact, in this case *bsc* will actually choose the representations 00, 01 an 10, respectively for these constants, but (a) that is an internal detail that need not concern us, and (b) BSV has mechanisms to choose alternate encodings if we so wish. The `Eq` part of line 2 tells the compiler define the `==` and the `!=` operators for this type, so we can compare to expressions of type `Color` for equality and inequality. The `FShow` part tells the compiler to define a canonical way to print these values in `$display` statements in symbolic form as the strings `"Red"`, `"Blue"` and `"Green"`, respectively.

The real payoff comes in strong type checking. Suppose, elsewhere, we have another type representing traffic light colors:

Enumerated types
```
typedef enum {Green, Yellow, Red} TrafficLightColor
        deriving (Bits, Eq, FShow);
```

These, too, will be represented using 00, 01 and 10, respectively, but strong typechecking will ensure that we never accidentally mix up the `Color` values with `TrafficLightColor` values, even though both are represented using 2 bits. Any attempt to compare values across these types, or to pass an argument of one of these type when the other is expected, will be caught as a static type checking error by *bsc*.

Processors execute instructions, which nowadays are often encoded in 32-bits. A few fields in a 32-bit instruction usually refer to ALU opcodes of a few classes. We define them using enumerated types like this:

Enumerated types for ALU opcodes

```
typedef enum {Eq, Neq, Le, Lt, Ge, Gt, AT, NT} BrFunc
        deriving (Bits, Eq);

typedef enum {Add, Sub, And, Or, Xor, Nor,
              Slt, Sltu, LShift, RShift, Sra} AluFunc
        deriving (Bits, Eq);
```

The first definition is for comparison operators, and the second one is for arithmetic, logic and shift operators.

2.4.3 Combinational ALUs

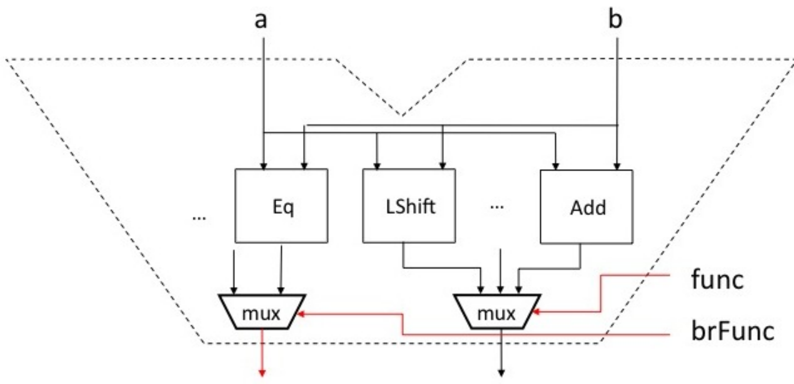


Figure 2.11: A combinational ALU

A combinational ALU can be regarded just as a composition of the various types and expressions we have seen thus far. The circuit is shown pictorially in Fig 2.11. In BSV code parts of this circuit are expressed using functions like the following, which can then be combined into the full ALU:

ALU partial function for Arith, Logic, Shift ops

```
function Data alu (Data a, Data b, AluFunc func);
  Data res = case(func)
     Add   : (a + b);
     Sub   : (a - b);
     And   : (a & b);
     Or    : (a | b);
     Xor   : (a ^ b);
     Nor   : ~(a | b);
     Slt   : zeroExtend( pack( signedLT(a, b) ) );
     Sltu  : zeroExtend( pack( a < b ) );
     LShift: (a << b[4:0]);
     RShift: (a >> b[4:0]);
     Sra   : signedShiftRight(a, b[4:0]);
  endcase;
  return res;
endfunction
```

ALU partial function for Comparison

```
function Bool aluBr (Data a, Data b, BrFunc brFunc);
  Bool brTaken = case(brFunc)
    Eq  : (a == b);
    Neq : (a != b);
    Le  : signedLE(a, 0);
    Lt  : signedLT(a, 0);
    Ge  : signedGE(a, 0);
    Gt  : signedGT(a, 0);
    AT  : True;
    NT  : False;
  endcase;
  return brTaken;
endfunction
```

2.4.4 Multiplication

We close this section with another small exercise, one of the arithmetic functions we may want in the ALU: multiplying an m-bit number and an n-bit number to produce an $m+n$-bit result. Note that some processors do not have a multiplier! They just implement a multiplication-by-repeated-addition algorithm in software. However, in most situations, where multiplication performance is important, one would want a hardware implementation.

Suppose we are multiplying two 4-bit values. Our high-school multiplication algorithm (translated from decimal to binary) looks like this:

Multiplication by repeated addition

```
            1 1 0 1    // Multiplicand, b
            1 0 1 1    // Multiplier,   a
            -------
            1 1 0 1    // b x a[0] (== b), shifted by 0
          1 1 0 1      // b x a[1] (== b), shifted by 1
        0 0 0 0        // b x a[2] (== 0), shifted by 2
      1 1 0 1          // b x a[3] (== b), shifted by 3
    ---------------
    1 0 0 0 1 1 1 1
```

Thus, the j^{th} partial sum is `(a[j]==0 ? 0 : b) << j`, and the overall sum is just a for-loop to add these sums. Fig. 2.12 illustrates the circuit, and here is the BSV code to express this:

BSV code for combinational multiplication

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
   Bit#(32) prod = 0;
   Bit#(32) tp = 0;
```

```
    for (Integer i=0; i<32; i=i+1) begin
       Bit#(32) m = (a[i]==0)? 0 : b;
       Bit#(33) sum = add32(m,tp,0);
       prod[i] = sum[0];
       tp = truncateLSB(sum);
    end
    return {tp,prod};
endfunction
```

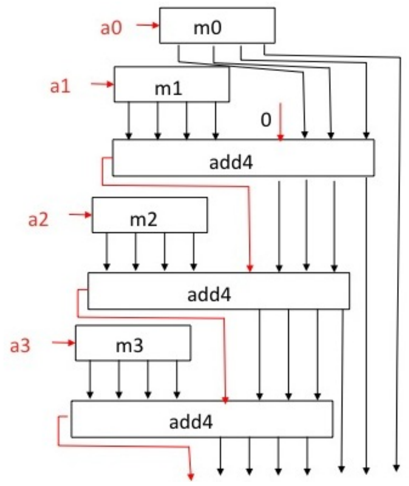


Figure 2.12: A combinational multiplier

2.5 Summary, and a word about efficient ALUs

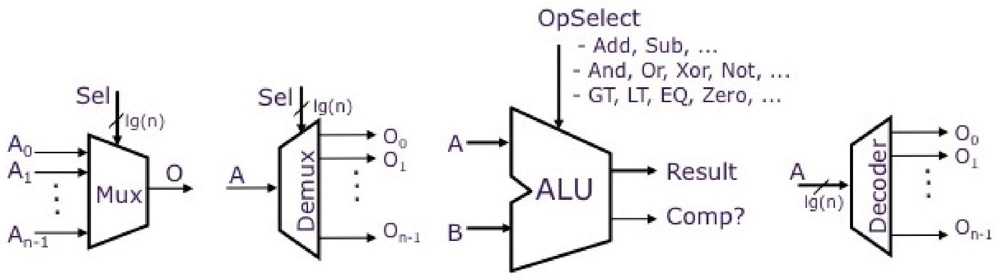


Figure 2.13: Various combinational circuits

Fig. 2.13 shows symbols for various combinational circuits. We have already discussed multiplexers and ALUs. A demultiplexer ("demux") transmits its input value to one of n output ports identified by the Sel input (the other output ports are typically forced to 0). A decoder takes an input A of $\log n$ bits carrying some value $0 \leq j \leq 2^n - 1$. It has n outputs such that the j^{th} output is 1 and all the other outputs are 0. We also say that the outputs represent a "one-hot" bit vector, because exactly one of the outputs is 1. Thus, a decoder is equivalent to a demux where the demux's Sel input is the decoder's A, and the demux's A input is the constant 1-bit value 1.

The simple examples in this chapter of a ripple-carry adder and a simple multiplier are not meant to stand as examples of efficient design. They are merely tutorial examples to demystify ALUs for the new student of computer architecture. Both our combinational ripple-carry adder and our combinational repeated-addition multiplier have very long chains of gates, and wider inputs and outputs make this worse. Long combinational paths, in turn, restrict the speed of the clocks of the circuits in which we embed these ALUs, and this ultimately affects processor speeds. We may have to work, instead, with *pipelined* multipliers that take multiple clock cycles to compute a result; we will discuss clocks, pipelines and so on in subsequent chapters.

In addition, modern processors also have hardware implementations for floating point operations, fixed point operations, transcendental functions, and more. A further concern for the modern processor designer is power consumption. Circuit design choices for the ALU can affect the power consumed per operation, and other controls may be able to switch off power to portions of the ALU that are not currently active. In fact, the topic of efficient computer arithmetic and ALUs has a deep and long history; people have devoted their careers to it, whole journals and conferences are devoted to it; here, we have barely scratched the surface.

Chapter 3

Sequential (Stateful) Circuits and Modules

3.1 Registers

3.1.1 Space and time

Recall the combinational "multiply" function from Sec. 2.4.4:

BSV code for combinational multiplication

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
   Bit#(32) prod = 0;
   Bit#(32) tp = 0;
   for (Integer i=0; i<32; i=i+1) begin
      Bit#(32) m = (a[i]==0)? 0 : b;
      Bit#(33) sum = add32(m,tp,0);
      prod[i] = sum[0];
      tp = truncateLSB(sum);
   end
   return {tp,prod};
endfunction
```

Static elaboration will unfold the loop into a combinational circuit that contains 32 instances of the `add32` circuit. The circuit is elaborated in *space*, i.e., it is eventually laid out in silicon on your ASIC or FPGA.

An alternative is to elaborate it in *time*: have only a single instance of the `add32` circuit which is used repeatedly 32 times in a temporal sequence of steps. To do this, we need a device called a "register" that will hold the partial results from one step so that they can be used as inputs to `add32` in the next step. We also use the words "state element" and "sequential element" for a register, because it is in different states over time.

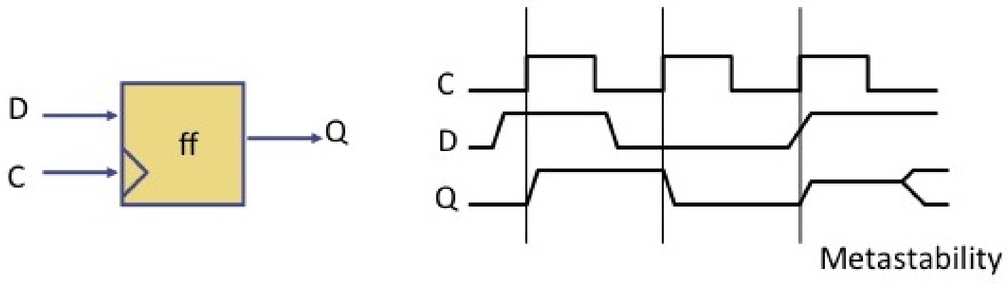


Figure 3.1: An edge-triggered D flip-flop and an example input-output waveform

3.1.2 D flip-flops

The left side of Fig. 3.1 shows the symbol for the most commonly used state element, an edge-triggered D flip-flop (DFF). Although the term "register" is more general, we typically use the word to refer to such a DFF. It has two inputs, D (data) and C (clock), and an output Q. The clock input C is typically a regular, square-wave oscillation. On each rising edge of C, the register samples the current value of D, and this value appears on the Q output "shortly" (needs to propagate from D to Q through internal circuits which typically have a small delay compared to the time period of C).

In the waveforms on the right side of Fig. 3.1, at the first rising edge of C, D has the value 1; the value of Q shortly becomes 1. At the next rising edge, D has value 0, and Q shortly becomes 0. The third rising edge illustrates a bad situation, where D is still changing at the rising edge. Here, it is possible for the DFF to enter the so-called "meta-stable" state where, like Hamlet, it vacillates over declaring the sampled value to be 0 or not 0 (i.e., 1). During this time, the Q output may not even be a valid digital value—its voltage may be in the "forbidden" range between the thresholds that we define as "0" and "1". As you can imagine, a downstream circuit does not get a clear, unambiguous 0 or 1, and so the problem can propagate through several levels of circuits. Worse, meta-stability may persist for an arbitrarily long time before it finally settles into the 0 or 1 state. For this reason, digital circuits are usually carefully designed to avoid this situation—clock periods are set long enough so that D inputs are always stable at 0 or 1 at the rising clock edge.

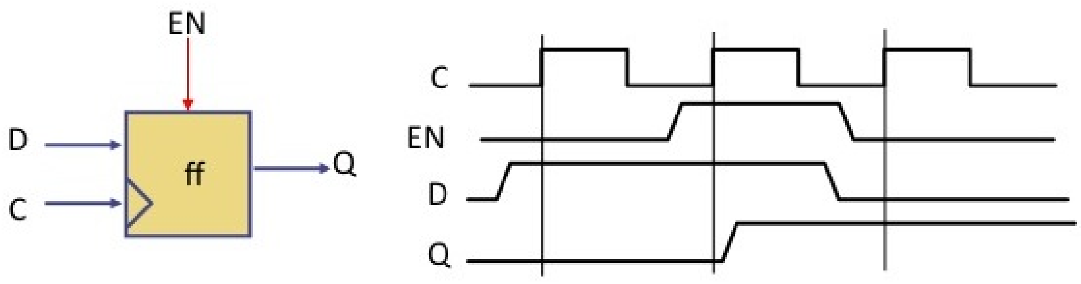


Figure 3.2: A D flip-flop with Write-Enable

D flip-flops often have an additional "Write Enable" input, illustrated in Fig. 3.2. Again, C is a continuous square-wave oscillation. However, the D input is sampled only when the EN input is true. Look at the example waveform on the right of the figure. Assume that Q is initially 0 (from some previous sampling of D). At the first rising clock edge, D is 1.

However, since EN is 0, it is not sampled, and Q retains its previous value of 0. At the second clock edge, EN is 1, and so the value of D (1) is sampled, and arrives at Q. At the third clock edge, EN is 0, so D (0) is not sampled, and Q remains at 1.

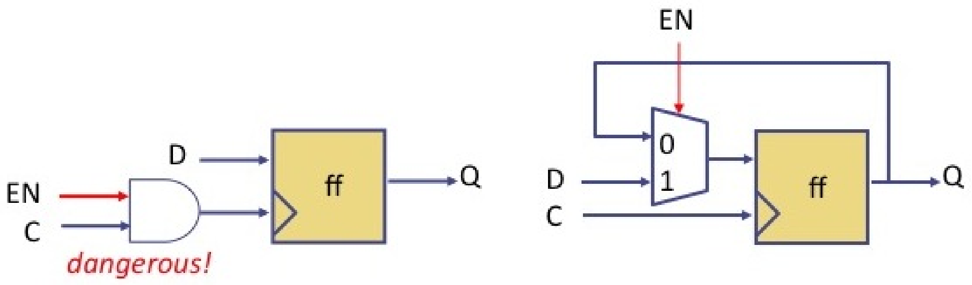


Figure 3.3: Implementing a D flip-flop with Write-Enable

Fig. 3.3 shows two attempts at implementing the Write-Enable feature using an ordinary D flip-flop. The attempt on the left may at first seem reasonable: just disable the clock when EN is 0, using an AND gate. But this is very dangerous, because the EN signal itself may be generated by some upstream circuit using the same clock C, and so it may change at roughly the same time as the clock edge. This reopens the door not only to metastability, but also to so-called "glitches", where a slight mismatch in the timing of the edges of EN and C can cause a momentary pulse to appear at the output of the AND gate, which may cause the DFF to sample the current D value. In general, in digital circuits, it is very dangerous to mess around with clocks as if they were ordinary logic signals. In large, complex, high-speed digital circuits, the clock signals are very carefully engineered by experienced specialists, and are kept entirely separate from the data paths of the circuit. The right side of Fig. 3.3 shows a safer way to implement the EN feature, where we do not tinker with the clock.

In BSV, clocks have a distinct data type `Clock`, and strong type checking ensures that we can never accidentally use clocks in ordinary computation expressions.

D flip-flops may also have a "Reset" signal which re-initializes the flip-flop to a fixed value, say 0. Reset signals may be synchronous (they take effect on the next rising clock edge) or asynchronous (they take effect immediately, irrespective of clock edges).

3.1.3 Registers

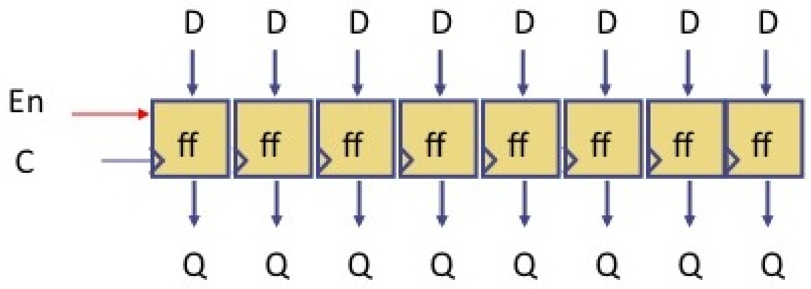


Figure 3.4: An n-bit register built with D flip-flops

Fig. 3.4 shows an n-bit "register" built with D flip-flops. All the clocks are tied together, as are all the EN signals, and we can think of the register as sampling an n-bit input value on the rising clock edge. In BSV, one declares and instantiates a register like this:

Declaring and Instantiating Registers
```
Reg #(Bit #(32))   s <- mkRegU;
Reg #(UInt #(32))  i <- mkReg (17);
```

The first line declares `s` to be a register that contains 32-bit values, with unspecified initial value ("U" for Uninitialized). The second line declares `i` to be a register containing a 32-bit unsigned integer with initial value 17 (when the circuit is reset, the register will contain 17). Registers are assigned inside rules and methods (we'll describe these shortly) using a special assignment syntax:[1]

Register assigment statements
```
    s <= s & 32'h000F000F;
    i <= i + 10;
```

In the first line, at a clock edge, the old value of register `s` (before the clock edge) is bitwise ANDed with a 32-bit hexadecimal constant value, and stored back into the register `s` (visible at the next clock edge). In the second line, `i` is incremented by 10.

In BSV, all registers are strongly typed. Thus, even though `s` and `i` both contain 32-bit values, the following assignment will result in a type checking error, because `Bit#(32)` and `UInt#(32)` are different types.

Type checking error on strongly typed registers
```
    s <= i;
```

BSV registers can hold any type, including data structures. Example:

Registers for non-numeric types
```
Reg #(EthernetHeader) ehdr <- mkRegU;
FIFOF #(EthernetPacket) input_queue <- mkFIFOF;
...
rule rl_foo;
   ehdr <= input_queue.first.header;
endrule
```

The first line declares and instantiates a register holding an Ethernet packet header. The second line declares and instantiates a FIFO containing Ethernet packets (we will discuss FIFOs in more detail later).

[1]This is the same as Verilog's "delayed assignment" syntax

3.2 Sequential loops with registers

Consider the following C code that repeatedly applies some function f to an initial value s_0, 32 times:

A small iteration example (C code)
```
    int s = s0;
    for (int i=0; i<32; i++)
        s = f (s);
```

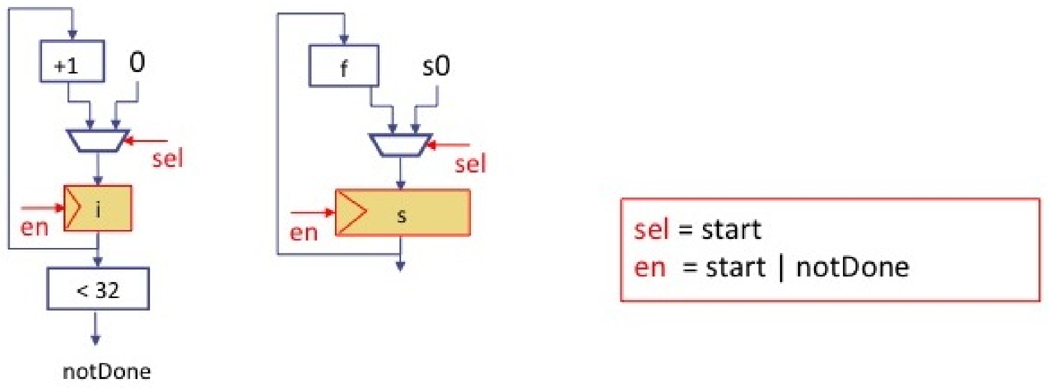


Figure 3.5: Circuit implementation of C iteration example

Fig. 3.5 shows a circuit that performs this function. It contains two registers holding `i` and `s`. When `start`, an external input signal pulse, is received, the muxes select the initial values `0` and `s0`, respectively; the EN signals are asserted, and the initial values are loaded into the registers. At this point, the `notDone` signal is true since `i` is indeed < 32. On subsequent clocks, the `i` register is incremented, and the `s` register is updated by `f(s)`. When `i` reaches 32, the `notDone` signal becomes false, and so the EN signals are no longer true, and there is no further activity in the circuit (until the next `start` pulse arrives). The following BSV code describes this circuit:

A small iteration example (BSV code)
```
1 Reg#(Bit#(32)) s <- mkRegU();
2 Reg#(Bit#(6))  i <- mkReg(32);
3
4 rule step if (i<32);
5    s <= f(s);
6    i <= i+1;
7 endrule
```

The rule on lines 4-7 has a name `step`, a rule condition or "guard" `(i<32)` and a rule body which is an "Action". The action itself consist of two sub-actions, `s <= f(s)` and `i <= i+1`.

It is best to think of a rule as an *instantaneous* action (zero time). A rule only executes if its condition is true. When it executes, all its actions happen simultaneously and instantaneously. Any value that is read from a register (such as `s` and `i` on the right-hand sides of the register assignments) will be the value prior to that instant. After the instant, the registers contain the values they were assigned. Since all the actions in a rule are simultaneous, the textual ordering between the above two actions is irrelevant; we could just as well have written:

Actions happen simultaneously (no textual order)
```
   i <= i+1;
   s <= f(s);
```

Thus, this rule acts as a sequential loop, repeatedly updating registers `i` and `s` until `(i>=32)`. When synthesized by *bsc*, we get the circuit of Fig.3.5.

A small BSV nuance regarding types

Why did we declare `i` to have the type `Bit#(6)` instead of `Bit#(5)`? It's because in the expression `(i<32)`, the literal value 32 needs 6 bits, and the comparision operator `<` expects both its operands to have the same type. If we had declared `i` to have type `Bit#(5)` we would have got a type checking error in the expression `(i<32)`.

What happens if we try changing the condition to `(i<=31)`, since the literal 31 only needs 5 bits?

Types subtlety
```
...
Reg#(Bit#(5))  i <- ...

rule step if (i<=31);
   ...
   i <= i+1;
endrule
```

This program will type check and run, but it will never terminate! The reason is that when `i` has the value 31 and we increment it to `i+1`, it will simply wrap around to the value 0. Thus, the condition `i<=31` is always true, and so the rule will fire forever. Note, the same issue could occur in a C program as well, but since we usually use 32-bit arithmetic in C programs, we rarely encounter it. We need to be much more sensitive to this in hardware designs, because we usually try to use the minimum number of bits adequate for the job.

3.3 Sequential version of the multiply operator

Looking at the combinational multiply operator code at the beginning of this chapter, we see that the values that change during the loop are `i`, `prod` and `tp`; we will need to hold these in registers when writing a sequential version of the loop:

BSV code for sequential multiplication
```
   Reg #(Bit#(32)) a    <- mkRegU;
   Reg #(Bit#(32)) b    <- mkRegU;
   Reg #(Bit#(32)) prod <- mkRegU;
   Reg #(Bit#(32)) tp   <- mkRegU;
   Reg #(Bit#(6))  i    <- mkReg (32);

   rule mulStep (i<32);
      Bit#(32) m   = ((a[i]==0)? 0 : b);
      Bit#(33) sum = add32(m,tp,0);
      prod[i] <= sum[0];              // (not quite kosher BSV)
      tp <= truncateLSB(sum);
      i <= i + 1;
   endrule
```

The register `i` is initialized to 32, which is the quiescent state of the circuit (no activity). To start the computation, something has to load `a` and `b` with the values to be multiplied and load `prod`, `tp` and `i` with their initial value of 0 (we will see this initialization later). Then, the rule fires repeatedly as long as `(i<32)`. The first four lines of the body of the rule are identical to the body of the for-loop in the combinational function at the start of this chapter, except that the assignments to `prod[i]` and `tp` have been changed to register assignments.

Although functionally ok, we can improve it to a more efficient circuit by eliminating dynamic indexing. Consider the expression `a[i]`. This is a 32-way mux: 32 1-bit input wires connected to the bits of `a`, with `i` as the mux selector. However, since `i` is a simple incrementing series, we could instead repeatedly shift `a` right by 1, and always look at the least significant bit (LSB), effectively looking at `a[0]`, `a[1]`, `a[2]`, ... Shifting by 1 requires no gates (it's just wires!), and so we eliminate a mux. Similarly, when we assign to `prod[i]`, we need a decoder to route the value into the i^{th} bit of `prod`. Instead, we could repeatedly shift `prod` right by 1, and insert the new bit at the most significant bit (MSB). This eliminates the decoder. Here is the fixed-up code:

BSV code for sequential multiplication

```
   Reg #(Bit#(32)) a    <- mkRegU;
   Reg #(Bit#(32)) b    <- mkRegU;
   Reg #(Bit#(32)) prod <- mkRegU;
   Reg #(Bit#(32)) tp   <- mkRegU;
   Reg #(Bit#(6))  i    <- mkReg (32);

   rule mulStep if (i<32);
      Bit#(32) m = ((a[0]==0)? 0 : b);  // only look at LSB of a
      a <= a >> 1;                      // shift a by 1
      Bit#(33) sum = add32(m,tp,0);
      prod <= { sum[0], prod [31:1] };  // shift prod by 1, insert at MSB
      tp <= truncateLSB(sum);
      i <= i + 1;
   endrule
```

Fig. 3.6 shows the circuit described by the BSV code. Compared to the combinational version,

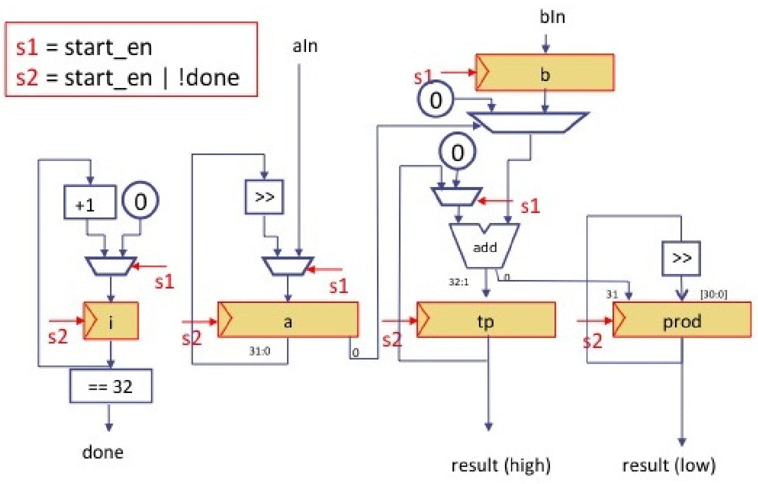


Figure 3.6: Sequential multiplication circuit

- The number of instances of the `add32` operator circuit has reduced from 32 to 1, but we have added some registers and muxes.
- The longest combinational path has been reduced from 32 cascaded `add32`s to one `add32` plus a few muxes (and so it can run at a higher clock speed).
- To complete each multiplication, the combinational version has a combinational delay corresponding to its longest path. The sequential version takes 32 clock cycles.

3.4 Modules and Interfaces

Just as, in object-oriented programming (OOP), we organize programs into classes/objects that interact via method calls, similarly in BSV we organize our designs into *modules* that interact using *methods*. And in BSV, just like OOP, we separate the concept of the *interface* of a module—"*What* methods does it implement and what are the argument and result types?"—from the module itself— "*How* does it implement the methods?".

For example, if we want to package our multiplier into a module that can be instantiated (reused) multiple times, we first think about the interface it should present to the external world. Here is a proposed interface:

Multiplier module interface

```
interface Multiply32;
   method Action    start  (Bit#(32) a, Bit#(32) b);
   method Bit#(64)  result;
endinterface
```

A module with this interface offers two methods that can be invoked from outside. The `start` method takes two arguments `a` and `b`, and has `Action` type, namely it just performs some action (this method does not return any result). In this case, it just kicks off the internal computation to multiply `a` and `b`, which may take many steps. The `result` method has no arguments, and returns a value of type `Bit#(64)`.

So far, this looks just like a conventional object-oriented language (with perhaps different syntax details). However, in BSV, a method is always invoked from a rule, either directly or indirectly from another method which in turn is invoked by a rule, and so on. In other words, all method invocations orginate in a rule. Just like a rule has a rule condition, a method may also have a method condition which limits when it may be invoked. This rule-based semantics is the key difference from traditional object-oriented languages. These rule-based semantics also have a direct hardware interpretation.

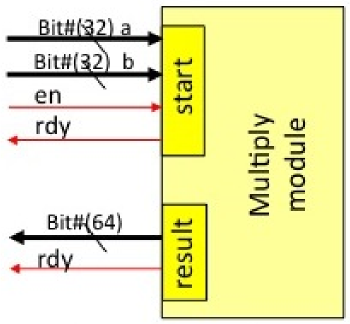


Figure 3.7: Hardware interpretation of interface methods

Fig. 3.7 illustrates how BSV methods are mapped into hardware. Every method is a bundle of input and output signals. Every method has one output signal called the "ready" signal (RDY) corresponding to the method condition. A rule in the external environment is allowed to invoke the method only when RDY is true. Method arguments become input signals (like `a` and `b`), and method results become output signals. Finally, `Action` methods like `start` have an input "enable" signal (EN) which, when asserted by the external rule, causes the method's action inside the module to happen.

We can now see how our multiplier is packaged into a module.

Multiplier module
```
module mkMultiply32 (Multiply32);
   Reg #(Bit#(32)) a    <- mkRegU;
   Reg #(Bit#(32)) b    <- mkRegU;
   Reg #(Bit#(32)) prod <- mkRegU;
   Reg #(Bit#(32)) tp   <- mkRegU;
   Reg #(Bit#(6))  i    <- mkReg (32);

   rule mulStep if (i<32);
      Bit#(32) m = ( (a[0]==0)? 0 : b );
      a <= a >> 1;
      Bit#(33) sum = add32(m,tp,0);
      prod <= { sum[0], prod [31:1] };
      tp <= truncateLSB(sum);
      i <= i + 1;
   endrule

   method Action start(Bit#(32) aIn, Bit#(32) bIn) if (i==32);
      a <= aIn;
```

```
19          b <= bIn;
20          i <= 0;
21          tp <= 0;
22          prod <= 0;
23       endmethod
24
25       method Bit#(64) result if (i == 32);
26          return {tp,prod};
27       endmethod
28    endmodule
```

In line 1, we declare the module `mkMultiply32` offering the `Multiply32` interface. Lines 2-15 are identical to what we developed in the last section. Lines 17-23 implement the `start` method. The method condition, or guard, is `(i==32)`. When invoked, it initializes all the registers. Lines 25-27 implement the `result` method. Its method condition or guard is also `(i==32)`. When invoked it returns `{tp,prod}` as its result.

BSV modules typically follow this organization: internal state (here, lines 2-6), followed by internal behavior (here, lines 8-15) followed by interface definitions (here, lines 17-27). As a programming convention, we typically write module names as `mk...`, and pronounce the first syllable as "make", reflecting the fact that a module can be instantiated multiple times.

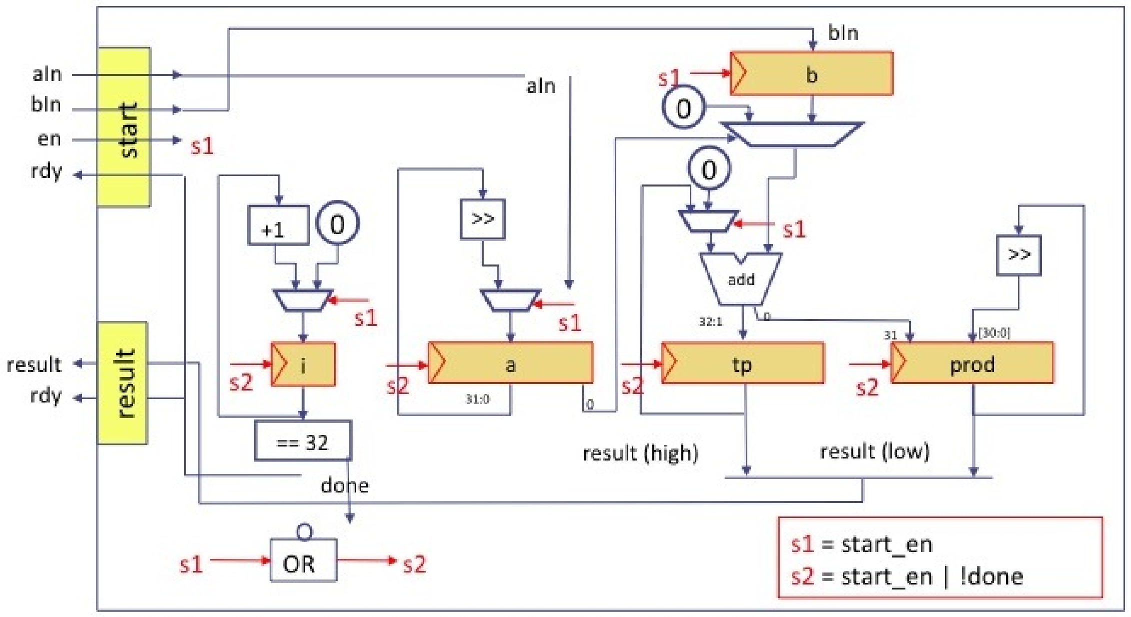


Figure 3.8: Multiplier module circuit

Fig. 3.8 shows the overall circuit of the module described by the BSV code. It pulls together all the individual pieces we have discussed so far.

To emphasize the distinction between an interface and a module that offers that interface, here is another module that implements exactly the same interface but uses our earlier combinational function.

Multiplier module, alternative

```
1  module mkMultiply32 (Multiply32);
2     Reg #(Bit#(64)) rg_result <- mkRegU;
3     Reg #(Bool)     done      <- mkReg (False);
4
5     function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
6        Bit#(32) prod = 0;
7        Bit#(32) tp = 0;
8        for (Integer i=0; i<32; i=i+1) begin
9           Bit#(32) m = ( (a[i]==0)? 0 : b );
10          Bit#(33) sum = add32(m,tp,0);
11          prod[i] = sum[0];
12          tp = truncateLSB(sum);
13       end
14       return {tp,prod};
15    endfunction
16
17    method Action start(Bit#(32) aIn, Bit#(32) bIn) if (! done);
18       rg_result <= mult32 (aIn, bIn);
19       done <= True;
20    endmethod
21
22    method Bit#(64) result if (done);
23       return rg_result;
24    endmethod
25 endmodule
```

Lines 4-14 are the same combinational function we saw earlier. The `start` method simply invokes the combinational function on its arguments and stores the output in the `rg_result` register. The `result` register simply returns the value in the register, when the computation is done.

Similarly, one can have many other implementations of the same multiplier interface, with different internal algorithms that offer various efficiency tradeoffs (area, power, latency, throughput):

Multiplier module, more alternatives

```
   module mkBlockMultiply (Multiply);
   module mkBoothMultiply (Multiply);
```

(”Block” multiplication and ”Booth” multiplication are two well-known algorithms, or circuit structures, but we do not explore them here.)

3.4.1 Polymorphic multiply module

Let us now generalize our multiplier circuit so that it doesn't just work with 32-bit arguments, producing a 64-bit result, but works with n-bit arguments and produces a $2n$-bit result. Thus, we could use the same module in other environments which may require, say, a 13-bit multiplier or a 24-bit multiplier. The interface declaration changes to the following:

Polymorphic multiplier module interface

```
interface Multiply #(numeric type tn);
   method Action             start  (Bit#(tn) a, Bit#(tn) b);
   method Bit#(TAdd#(tn,tn)) result;
endinterface
```

And the module definition changes to the following:

Polymorphic multiplier module

```
module mkMultiply (Multiply #(tn));
   Integer n = valueOf (tn);

   Reg #(Bit#(tn)) a    <- mkRegU;
   Reg #(Bit#(tn)) b    <- mkRegU;
   Reg #(Bit#(tn)) prod <- mkRegU;
   Reg #(Bit#(tn)) tp   <- mkRegU;
   Reg #(Bit#(TAdd#(1,TLog#(tn)))) i <- mkReg (fromInteger(n));

   rule mulStep if (i<32);
      Bit#(tn) m   = (a[0]==0)? 0 : b;
      a <= a >> 1;
      Bit#(TAdd#(tn,1)) sum = addN(m,tp,0);
      prod <= { sum[0], prod [n-1:1] };
      tp <= truncateLSB(sum);
      i <= i + 1;
   endrule

   method Action start(Bit#(tn) aIn, Bit#(tn) bIn) if (i==fromInteger(n));
      a <= aIn;
      b <= bIn;
      i <= 0;
      tp <= 0;
      prod <= 0;
   endmethod

   method Bit#(64) result if (i==fromInteger(n));
      return {tp,prod};
   endmethod
endmodule
```

Note the use of the pseudo-function `valueOf` to convert from an integer in the type domain to an integer in the value domain; the use of the function `fromInteger` to convert from type `Integer` to type `Bit#(...)`, and the use of type-domain operators like `TAdd`, `TLog` and so on to derive related numeric types.

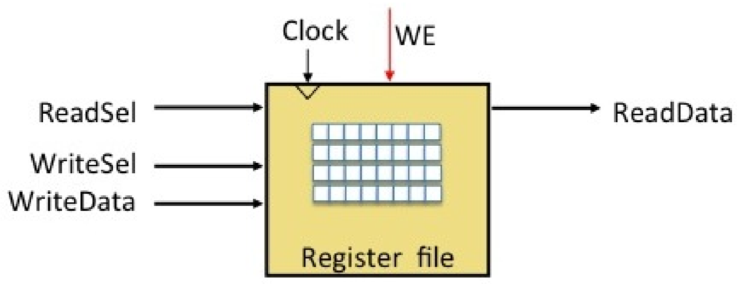


Figure 3.9: A Register File with four 8-bit registers

3.5 Register files

Registers files are arrays of registers, with a way to selectively read and write an individual register indentified by an index. Fig. 3.9 illustrates such a circuit component. The ReadSel signal is an index that identifies one of the n registers in the register file. The data in that register is presented on the ReadData output. This is usually a combinational function, just like reading a single register. The WriteSel signal identifies one of the registers, and if the WE enable signal is true, then WriteData is stored into that selected register.

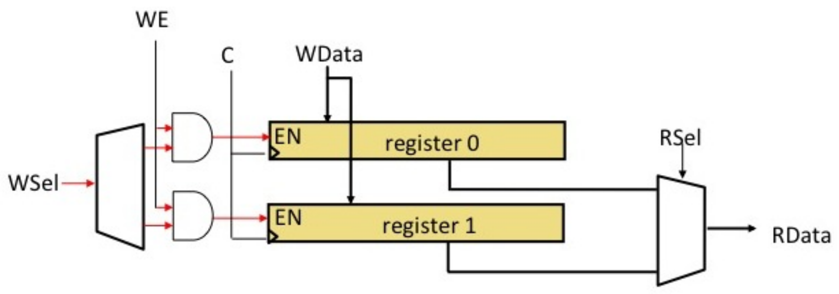


Figure 3.10: Register File implementation

Fig. 3.10 illustrates how a register file can be implemented. The read and write circuits are entirely separate. The read circuit is just a mux choosing the Q outputs of one of the registers. In the write circuit, Write Data is fed to the D inputs of all the registers. The Write Enable signal is fed to only one selected register, which captures the Write Data.

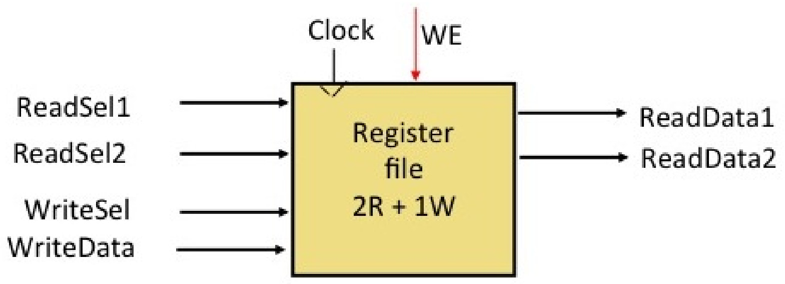


Figure 3.11: Multi-port Register File

The read and write interfaces of the register file are called "ports". Many register files have more than one read and/or write port. Fig. 3.11 illustrates a register file with two read ports and one write port. It is easy to see how to extend the implementation of Fig. 3.10 for this—simply replicate the mux that implements the read circuit. When implementing multiple write ports, one has to define what it means if both Write Selects are the same, i.e., both ports want to write to the same register, possibly with different values. We may declare this illegal, or a no-op, or fix a priority so that one of them is ignored. All these schemes are easy to implement.

Why are multi-port register files interesting? In a CPU, the architectural register set may be implemented directly as a register file. Many instructions (such as ADD) read two registers, perform an op, and write the result to a register. If we wish to execute one such instruction on every clock cycle, then clearly we need a 2-read port, 1-write port register file. If we wish to execute two such instructions in every clock (as happens is modern "superscalar" designs), we'll need a 4-read port, 2-write port register file.

Register files are declared and instantiated in BSV like this:

Declaring and Instantiating Register Files

```
import RegFile :: *;
...
module ...
   ...
   RegFile #(Bit#(5), Bit#(32)) rf1 <- mkRegFileFull;
   RegFile #(Bit#(5), Bit#(32)) rf2 <- mkRegFile (1,31);
   ...
endmodule
```

The first line is typically found at the top of a source file, and imports the BSV library package for register files. The `RegFile` declaration lines (found inside a module in the file) declares `rf1` to be a register file that is indexed by a 5-bit value (and so can contain $2^5 = 32$ registers), and whose registers contain 32-bit values. The `mkRegFileFull` module creates a full complement of registers for the 5-bit index, i.e., 32 registers. The next line is similar, except that we only allocate 31 registers, indexed from 1 to 31 (for example, in many architectures, register 0 is defined as a "constant 0" register, and so we may choose not to have a physical register for it).

Register files are accessed using the `sel` and `upd` method. The following example shows reading from register 2 and 5, adding them, and storing the result back in register 13.

Using a register file

```
   rule rl_add_instruction;
      let arg1 = rf2.sel (2);
      let arg2 = rf2.sel (5);
      rf2.upd (13, arg1 + arg2)
   endrule
```

3.6 Memories and BRAMs

A register file is essentially a small "memory". The Read Select or Write Select input is an address identifying one location in the memory, and that location is read and/or written.

However, building large memories out of register files is expensive in silicon area and power consumption. It is rare to see register files with more than a few dozen locations.

Typically, larger memories are built with special primitives that have been carefully engineered to be very dense (low area) and requiring much less power. The next step up in size from register files are on-chip SRAMs (Static RAMs), typically holding kilobytes to a few megabytes. They typically do not have more than one or two ports. The BSV library contains a number of SRAM modules which you get when you import the `BRAM` package:

The BRAM package
```
import BRAM :: *;
```

When you synthesize your BSV code for FPGAs, these will typically be "inferred" by downstream FPGA synthesis tools as "Block RAMs" which are built-in high-density SRAM structures on FPGAs. Whey you synthesize for ASICs, these will typically be "inferred" as SRAMs by downstream ASIC synthesis tools.

For even larger memories, you typically have to go to off-chip SRAMs (multi-megabytes) and DRAMs (Dynamic RAMs) in the megabyte to gigabyte range. On chip, you will typically have a "memory interface" circuit that communicates with the off-chip memories.

Unlike register files, SRAMs and DRAMs usually do not have combinational reads, i.e., data is only available 1 or more clock cycles after the address has been presented. And they typically do not have multiple ports for reading or writing.

Chapter 4

Pipelining Complex Combinational Circuits

4.1 Introduction

In the last chapter we introduced sequential elements (registers), motivated by “folding” a computation from an elaboration in space (combinational circuit) to an elaboration in time, which reduced area. Let us now consider the *throughput* or *repetition rate* of such circuits. Suppose the circuit computes some function $f()$ (e.g., multiply), and we wish repeatedly to compute $f()$ on a series of input datasets as fast as possible. The throughput of a circuit is measured by how often we can feed successive sets of inputs (which, in steady state, will be the same as how often we can extract successive sets of outputs). Throughput is often expressed in units such as as samples/second, megabits/second, etc., or by its inverse, the repetition rate, such as nanoseconds/sample.

In the pure combinational version (elaborated in space), we can feed the next set of inputs to the circuit only after the current set has made it safely through the circuit, i.e., after the longest combinational delay through the circuit. In practice what this means is that the combinational circuit is likely to have registers at either end, and we will only clock these registers at a rate that ensures that signals have propagated safely through the combinational circuit.

In the folded version (elaborated in time), after supplying one set of inputs we must clock the circuit n times where n is the degree of folding, producing the final answer, before we supply the next set of inputs. The folded version can likely be clocked at much higher speeds than the pure combinational version, since the largest combinational delay is likely to be much smaller. However, the maximum throughput is likely to be somewhat less than the pure combinational version because, in both versions, each sample must flow through the same total amount of combinational logic before the next sample is accepted, and in the folded version there is likely to some extra timing margin for safe registering in each circulation of data.

Note that in both the above versions, at any given time, the entire circuit is only involved in computation for one set of inputs, i.e., the output must be delivered before we can accept the next set of inputs.

In this chapter, we look at using registers purely for improving throughput via *pipelining*. We will insert registers in the combinational version, and hence area will increase; however throughput will also increase because earlier stages of the circuit can safely start on the next set of inputs while later stages are still working on previous sets of inputs.

4.2 Pipeline registers and Inelastic Pipelines

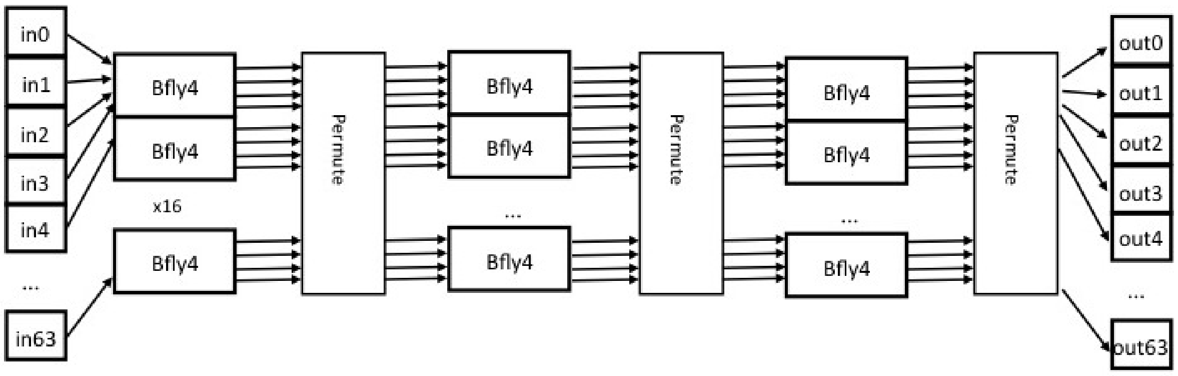


Figure 4.1: Combinational Inverse Fast Fourier Transform (IFFT)

Fig. 4.1 is a sketch of a combinational circuit for the Inverse Fast Fourier Transform function which is widely used in signal processing circuits, including the ubiquitous WiFi wireless networks. For the purposes of the current discussion the internal details of the blocks shown are not important but, briefly, the input is a set of 64 complex numbers; each "butterfly 4" block (Bfly4) performs some complex arithmetic function on 4 complex numbers, producing 4 output complex numbers; and each "Permute" block permutes its 64 input complex numbers into its output 64 complex numbers. Note that the longest combinational path will go through 3 Bfly4 blocks and 3 Permute blocks.

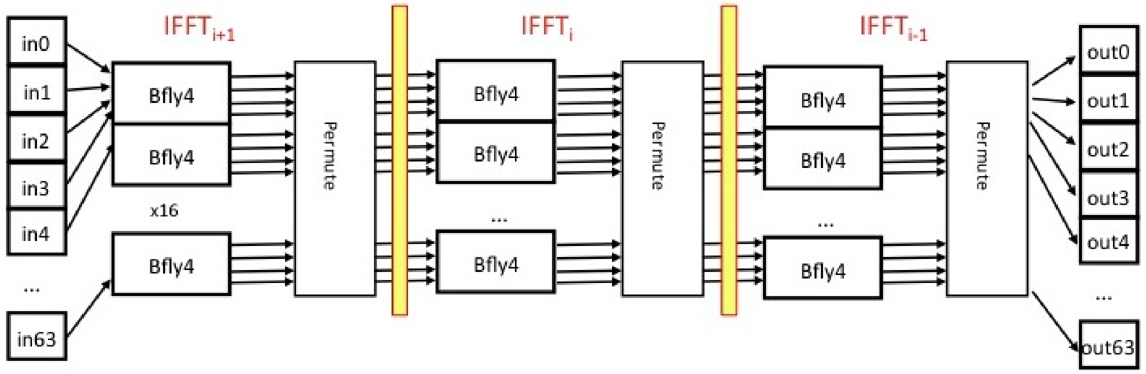


Figure 4.2: Pipelined Inverse Fast Fourier Transform (IFFT)

Fig. 4.2 shows the same circuit after the insertion of "pipeline registers" after the first two Permute blocks. We say that the circuit has been pipelined into three "stages". On the first clock edge we present the first set of inputs, which propagates through the Bfly4s and Permute block during the subsequent clock cycle. On the second clock edge, the first register safely captures this intermediate result, and we can immediately present the second set of inputs. During the subsequent clock cycle, the first stage works on the second set of inputs while the second stage works on the first set of inputs. In steady state, as shown by the labels at the top of the diagram, the three stages are working on input sets $i+1$, i and $i-1$, respectively. In particular, whereas a combinational or folded circuit works on one input set at a time, this pipelined circuit works in parallel on three sets of inputs.

4.2.1 Inelastic Pipelines

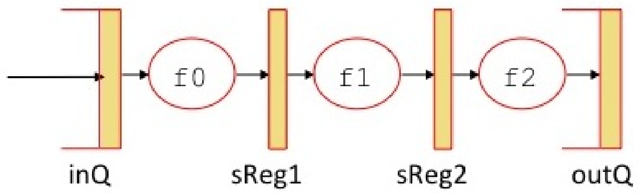


Figure 4.3: Inelastic pipeline for IFFT

Fig. 4.3 shows an abstraction of the IFFT circuit where each f_j represents a stage computation (column of Bfly4 boxes and a Permute box). InQ and OutQ represent input and output queues or FIFOs (we will discuss FIFOs in more detail in Sec. 4.3). The following code expresses the behavior of the pipeline:

Pipeline behavior

```
1  rule sync_pipeline;
2    sReg1 <= f0 (inQ.first());    inQ.deq();
3    sReg2 <= f1 (sReg1);
4    outQ.enq (f2 (sReg2));
5  endrule
```

In line 2, we apply f_0 to the head of the input queue and capture it in register `sReg1`; and we also remove the element from the queue. In line 3, we apply f_1 to the previous value sitting in `sReg1` and capture its result in `sReg2`. In line 4, we apply f_2 to the previous value sitting in `sReg2` and enqueue its result on the output queue.

When can this rule fire, and actually perform these actions? Our FIFOs will typically have conditions on their methods such that `first` and `deq` cannot be invoked on empty FIFOs, and `enq` cannot be invoked on full FIFOs. Thus, the rule will only fire, and move data forward through the pipe when `inQ` is not empty and `outQ` is not full. The rule could have been written with the conditions made explicit (redundantly) as follows:

Pipeline behavior with conditions made explicit

```
1  rule sync_pipeline (!inQ.empty() & !outQ.full);
2    sReg1 <= f0 (inQ.first());    inQ.deq();
3    sReg2 <= f1 (sReg1);
4    outQ.enq (f2 (sReg2));
5  endrule
```

Recall that, semantically, all the actions in a rule occur simultaneously and instantaneously. We also say that a rule is an *atomic* action or transaction. The code within a rule should never be read as a sequential program. For example, there is no chance that, after dequeueing a dataset from `inQ`, we somehow then find that we are unable to enqueue into `outQ`. Either the rule fires and everything in the rule happens, or it stalls and nothing happens. As will become evident through the rest of the book, this is a key and powerful property for reasoning about the correctness of circuits. It is what fundamentally distinguishes BSV from other hardware design languages.

4.2.2 Stalling and Bubbles

Suppose `outQ` is not full, but `inQ` is empty. The rule cannot fire, as just discussed, and we say that the pipeline is *stalled*. Unfortunately, also, any data sitting in `sReg1` and `sReg2` will be stuck there until fresh input data arrive allowing the pipeline to move. We'll now modify the rule to permit such data to continue moving. Once we do this, of course, we have to distinguish the situation when a pipeline register actually contains data *vs.* it is "empty". We do this by adding a flag bit to each pipeline register to indicate whether it contains valid data or not, as illustrated in Fig. 4.4.

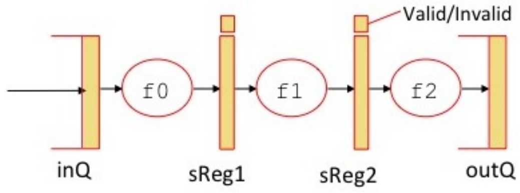


Figure 4.4: Valid bits on pipeline registers

When a pipeline register is empty, we also call this a *bubble* in the pipeline. The modified code is shown below.

Adding Valid bits

```
1   rule sync_pipeline;
2      if (inQ.notEmpty) begin
3         sReg1  <= f0 (inQ.first); inQ.deq;
4         sReg1f <= Valid
5      end
6      else
7         sReg1f <= Invalid;
8
9      sReg2  <= f1 (sReg1);
10     sReg2f <= sReg1f;
11
12     if (sReg2f == Valid)
13        outQ.enq (f2 (sReg2));
14  endrule
```

Lines 2-7 are the first stage: if `inQ` has data, we move the data as usual into `sReg1`, and also set `sReg1f` to the constant Valid. If `inQ` has no data, we set `sReg1f` to Invalid.

Lines 9-10 are the second stage: we move the data through f_1, and copy the Valid state from `sReg1f` to `sReg2f`. Note that if `sReg1f` was Invalid, then line 9 is reading invalid data from `sReg1` and therefore storing invalid data in `sReg2`, but it doesn't matter, since the Invalid tag accompanies it anyway. Finally, lines 12-13 are the third stage; we perform f_3 and enqueue an output only if `sReg2f` is Valid.

Under what conditions will this rule fire? Recall that `inQ.first` and `inQ.deq` are enabled only if `inQ` has data. However, *bsc* will recognize that this only matters if `inQ.notEmpty` is true. Thus, the empty/full state of `inQ` does not prevent the rule from firing: if empty, we don't attempt `.first` or `.deq` and only execute the `else` clause of the condition; if full, we do `.first` and `.deq` etc.

Similarly, recall that `outQ.enq` is enabled only if `outQ` is not full. However, we only attempt this if `sReg2f` is Valid. Thus, the rule will only stall if we actually have data in `sReg2` and `outQ` is full. The following table shows all the possible combinations of the two valid bits, `inQ` being empty or not, and `outQ` being full or not.

`inQ`	`sReg1f`	`sReg2f`	`outQ`	Rule fires?
notEmpty	Valid	Valid	notFull	fire
notEmpty	Valid	Valid	full	stall
notEmpty	Valid	Invalid	notFull	fire
notEmpty	Valid	Invalid	full	fire
notEmpty	Invalid	Valid	notFull	fire
notEmpty	Invalid	Valid	full	stall
notEmpty	Invalid	Invalid	notFull	fire
notEmpty	Invalid	Invalid	full	fire
empty	Valid	Valid	notFull	fire
empty	Valid	Valid	full	stall
empty	Valid	Invalid	notFull	fire
empty	Valid	Invalid	full	fire
empty	Invalid	Valid	notFull	fire
empty	Invalid	Valid	full	stall
empty	Invalid	Invalid	notFull	fire
empty	Invalid	Invalid	full	fire

4.2.3 Expressing data validity using the Maybe type

In this section we take a small excursion into modern language technology. There are no new hardware structures here; we just show how BSV exploits modern strong type checking for *safe* expression of hardware structures.

In the last code fragment, recall that the action `sReg2<=f1(sReg1)` may be computing with invalid data. Fortunately in this case it does not matter, since we are also carrying across the accompanying bit `sReg2f`; if `sReg1` was invalid, then the invalid data in `sReg2` is also marked invalid. There are many situations were we might not be so lucky, for example if the f_1 computation itself involved changing some state or raising an exception, such as a divide-by-zero. To avoid such problems, we should always guard computations on potentially invalid data with a test for validity.

In BSV we can rely on strong static type checking to enforce this discipline. The standard idiom is the `Maybe` type which is declared as follows:

The Maybe type
```
typedef union tagged {
   void    Invalid;
   data_t  Valid;
} Maybe #(type data_t)
  deriving (Bits, Eq);
```

A BSV tagged union type, similar to a `union` type in in C/C++, is an "either-or": either it is `Invalid`, in which case there is no data associated with it (`void`), or it is `Valid`, in which case it has some data of type `data_t` associated with it. As in the examples in Sec. 2.4.2, the last line is a standard BSV incantation to tell *bsc* to choose a canonical representation in bits, and to define the `==` and `!=` operators for this type.

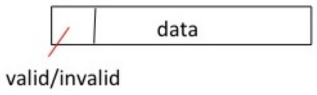


Figure 4.5: Representation in bits of values of Maybe type

Fig. 4.5 shows how, if a value of type `t` is represented in n bits, then a value of `Maybe#(t)` can be represented in $n+1$ bits. The extra bit represents the Valid/Invalid tag, and is placed at the MSB end (most significant bit). The representation *always* takes $n+1$ bits, whether Valid or Invalid; in the latter case, the lower n bits are unpredictable (effectively garbage). Note that we show this representation here only to build intuition; in practice we never think in terms of representations, just in terms of abstract `Maybe` types. This insulates us from having to re-examine and change a lot of distributed source code if, for example, we decide on a change in representation.

We construct a `Maybe`-type value using expressions like this:

Maybe value construction
```
   tagged Invalid    // To construct an Invalid value
   tagged Valid x    // To construct a Valid value with data x
```

We can examine `Maybe`-type values using the following functions:

BSV code
```
    isValid (mv)       // To test if a Maybe value is Valid or not
    fromMaybe (d, mv)  // To extract a data value from a Maybe value
```

The `isValid()` function is applied to a Maybe value and returns boolean True if it is Valid and False if Invalid. The `fromMaybe()` function is applied to a default value `d` of type `t` and a `Maybe#(t)` value `mv`. If `mv` is Valid, the associated data is returned; if Invalid, then `d` is returned. Notice that in no circumstances do we ever access the garbage data in an Invalid Maybe value. This is precisely what makes tagged unions in BSV (and in languages like Haskell, ML and so on) fundamentally different from ordinary unions in C/C++ which are not type safe.

Let us now rewrite our `sync_pipeline` rule using Maybe types:

Using Maybe types

```
1  module ...
2     ...
3     Reg #(Maybe #(t)) sReg1 <- mkRegU;
4     Reg #(Maybe #(t)) sReg2 <- mkRegU;
5
6     rule sync_pipeline;
7        if (inQ.notEmpty)
8           sReg1  <= tagged Valid (f0 (inQ.first)); inQ.deq;
9        else
10          sReg1 <= tagged Invalid;
11
12       sReg2 <= ( isValid (sReg1) ? tagged Valid f1 (fromMaybe (?, sReg1))
13                                  : tagged Invalid );
14
15       if (isValid (sReg2))
16          outQ.enq (f2 (fromMaybe (?, sReg2)));
17    endrule
18 ...
19 endmodule
```

In lines 3-4 we declare `sReg1` and `sReg2` now to contain Maybe values. In lines 8 and 10 we construct Valid and Invalid values, respectively. In lines 12-13 we test `sReg1` for validity, and we construct Valid and Invalid values for `sReg2`, respectively. We supply a "don't care" argument (`?`) to `fromMaybe` because we have guarded this with a test for validity. Finally, in lines 15-16 we test `sReg2` for validity, and use `fromMaybe` again to extract its value.

Instead of using the functions `isValid` and `fromMaybe`, tagged unions are more often examined using the much more readable "pattern matching" notation:

Using Maybe types

```
   rule sync_pipeline;
      if (inQ.notEmpty)
         sReg1  <= tagged Valid (f0 (inQ.first)); inQ.deq;
      else
         sReg1 <= tagged Invalid;

      case (sReg1) matches
         tagged Invalid : sReg2 <= tagged Invalid;
         tagged Valid .x: sReg2 <= tagged Valid f1 (x);
      endcase

      case (sReg2) matches
         tagged Valid .x: outQ.enq (f2 (x));
      endcase
   endrule
```

Notice the keyword `matches` in the case statements. In each clause of the case statements, the left-hand side (before the ":") is a pattern to match against the Maybe value. In line 9, the pattern `tagged Valid .x` not only checks if `sReg1` is a valid value, but it also binds the new variable `x` to the contained data; this is used in the right-hand side as an argument to `f1`. Note the dot "`.`" in front of the `x` in the pattern: this signals that `x` is a new pattern variable that is effectively declared at this point, and must be bound to whatever is in the value in that position (its type is obvious, and inferred, based on its position in the pattern). The scope of `x` is the right-hand side of that case clause.

4.3 Elastic Pipelines with FIFOs between stages

In inelastic pipelines (previous section) each stage is separated by an ordinary register, and the overall behavior is expressed in a rule that combines the behavior of each stage into a single atomic (and therefore synchronous) transaction. In elastic pipelines, each stage is separated by a FIFO, and each stage's behavior is expressed as rule that may fire independently (asynchronously) with respect to the other stages.

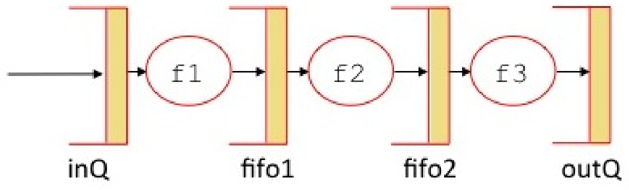


Figure 4.6: An elastic pipeline

Fig. 4.6 shows a sketch of an elastic pipeline. The BSV code structure is shown below:

Elastic pipeline
```
import FIFOF :: *;

module ...
   ...
   FIFOF #(t) inQ   <- mkFIFOF;
   FIFOF #(t) fifo1 <- mkFIFOF;
   FIFOF #(t) fifo2 <- mkFIFOF;
   FIFOF #(t) outQ  <- mkFIFOF;

   rule stage1;
      fifo1.enq (f1 (inQ.first);  inQ.deq;
   endrule

   rule stage2;
      fifo2.enq (f2 (fifo1.first); fifo1.deq();
   endrule

   rule stage3;
      outQ.enq (f3 (fifo2.first); fifo2.deq;
   endrule
   ...
endmodule
```

Line 1 shows how we import the BSV `FIFOF` package to make FIFOs available. Lines 5-8 show typical instantiations of FIFO modules. This is followed by three rules, each of which dequeues a value from an upstream FIFO, performs a compuation on the value and enqueues the result into a downstream FIFO. Each rule can fire whenever its upstream FIFO is not empty and its downstream FIFO is not full. The following table shows some of the possible situations, where NE and NF stand for "not empty" and "not full", respectively. Note that any FIFO that has capacity > 1 can be simultaneously not empty and not full.

`inQ`	`fifo1`	`fifo2`	`outQ`	`stage1`	`stage2`	`stage3`
NE	NE,NF	NE,NF	NF	Yes	Yes	Yes
NE	NE,NF	NE,NF	F	Yes	Yes	No
NE	NE,NF	NE,F	NF	Yes	No	Yes
NE	NE,NF	NE,F	F	Yes	No	No
...	...	...	...	...	...	...

Let us revisit the question of bubbles and stalls in this context. Suppose, initially all FIFOs are empty. Then, none of the rules can fire, and the circuit lies quiescent. Now suppose a value arrives in `inQ`. Now, `stage1` can fire, advancing the value into `fifo1`; then, `stage2` can fire, advancing it to `fifo2`; finally, `stage3` can fire, advancing it to `outQ`. Thus, the independent firing of rules allows values to flow through the pipeline whenever the FIFO ahead is not full; values never get stuck in the pipeline. Each stage can stall individually. Also note that we only ever compute and move data when an actual value is available, so there is no need for Maybe types and values here. Another way of looking at this is that the Valid bits have effectively become full/empty bits inside the FIFOs.

What happens if, initially, `inQ` contains value x followed by value y? As before, `stage1` can fire, advancing $f_1(x)$ into `fifo1`, and then `stage2` can then advance $f_2(f_1(x))$ into `fifo2`. But, during this second step, is it possible for `stage1` concurrently to advance $f_1(y)$ from `inQ` into `stage1`? For this to happen, `fifo1` must permit its `enq` method to be invoked concurrently with its `first` and `deq` methods. If not, `stage1` and `stage2` could not fire concurrently; and `stage2` and `stage3` could not fire concurrently. Adjacent stages could at best fire in alternate steps and we would hardly get what we think of as pipelined behavior! Note that it would still produce functionally correct results $(f_3(f_2(f_1(x))))$ for each input x; but it would not meet our performance goals (throughput).

The topic of concurrent method invocation is a deep one because one needs to define the semantics carefully. If method m_1 is invoked before method m_2 for a module M, it is easy to define what this means: m_1 peforms a transformation of M's state, and that final state is the initial state for m_2, and so on. But when m_1 and m_2 are invoked "concurrently", there are often choices about what this means. For example, if `enq` and `deq` are invoked concurrently on a FIFO with 1 element, are we enqueuing into a FIFO with 1 element or into an empty FIFO (because of the `deq`)? Neither choice is inherently right or wrong; it is a genuine option in semantic definition. BSV semantics provide a very precise vocabulary in which to specify our choices, and BSV is expressive enough to permit implementing any such choice. We shall study this topic in detail in the next chapter.

4.4 Final comments on Inelastic and Elastic Pipelines

Inelastic pipeline stages are typically separated by ordinary registers, and behavior (data movement) is expressed as a single rule that specifies the actions of all the stages. All the actions happen as a single, atomic transaction. We saw that, in order to handle pipeline bubbles and keep data moving, we had to introduce a lot of "flow control" control logic: valid bits to accompany data, and many conditionals in the rule to manage data movement. It was not trivial to reason about correctness, i.e., that our circuit did not accidentally drop a valid sample or overwrite one with another. The problem with writing pipelines in this style (which is the natural style in Verilog and VHDL) is that it does not evolve or scale easily; it gets more and more complex with each change, and as we go to larger pipelines. For scalability one would like a *compositional* way to define pipelines, where we define stages individually and then compose them into pipelines.

Elastic pipeline stages are typically separated by FIFOs, and each stage's behavior is expressed as an individual, local rule. Flow control is naturally encapsulated into method conditions preventing us from enqueuing into a full FIFO and from dequeuing from an empty FIFO. Correctness is more obvious in such pipelines, and they are easier to write in a compositional (modular) way. However, when we compose elastic descriptions into larger systems, we must ensure that rules can fire concurrently when they have to, so that we achieve our desired throughput.

4.5 Variations on architecture for IFFT

In this chapter, so far, we have seen how the combinational IFFT can be pipelined by the addition of pipeline registers after various stages in the combinational circuit. This adds area (because of the pipeline registers) but improves throughput because successive data samples can now be streamed through at a higher frequency.

In the last chapter, we saw how a repeated computation can folded in space by circulating data repeatedly through a single instance of the computation. This reduces area (we only have a single instance) but does not improve throughput. This principle can be applied to IFFT, since the `stage_f` computation is repeated three times. In fact, this principle can be applied further: in each stage, there are sixteen Bfly4 computations for which we can use a single circuit instance through which we sequentially pass the sixteen inputs to those computations. We call this a "super-folded" architecture.

In this final section of this chapter, we sketch the code for all these variations. This should reinforce the idea that, for the same mathematical function (IFFT) we can implement a range of architectures that may vary widely in latency, throughput, area and power consumption. Which one is "best" depends on external requirements; different applications may place different priorities on those measures.

4.5.1 Combinational IFFT

The fully combinational version of IFFT can be expressed in the following BSV code:

Combinational IFFT

```
1  function Vector#(64, Complex#(n)) ifft
2                                    (Vector#(64, Complex#(n)) in_data);
3     Vector#(4,Vector#(64, Complex#(n))) stage_data;
4
5     stage_data[0] = in_data;
6
7     for (Bit#(2) stage = 0; stage < 3; stage = stage + 1)
8        stage_data[stage+1] = stage_f (stage, stage_data[stage]);
9
10    return(stage_data[3]);
11 endfunction
```

The `for`-loop in lines 7-8 will be statically unfolded to create a composition of three instances of the `stage_f` function, where each stage is a column of sixteen Bfly4s followed by a Permute (we omit the internal details of `stage_f` here).

4.5.2 Pipelined IFFT

The elastic pipelined version of IFFT is just an application of the principles of 4.3:

Elastic pipelined IFFT

```
1  import FIFOF :: *;
2
3  module ...
4     ...
5     Vector #(4, FIFOF #(Vector#(64, Complex#(n)))) fifos
6                                                  <- replicateM (mkFIFOF);
7
8     for (Bit #(2) stage = 0; stage < 3; stage = stage + 1)
9        rule stage;
10          fifo[j+1].enq (stage_f (stage, fifo[j].first);  fifo[j].deq;
11       endrule
12    ...
13 endmodule
```

In line 5 we declare a vector of four FIFO interfaces, and in line 6 we populate it by instantiating four FIFOs. The `replicateM` function takes a module (in this case `mkFIFOF` and instantiates it repeatedly to return a vector of interfaces. The number of instances is specified by the size of the required vector type, in this case, 4. The for-loop in lines 8-11 will be statically elaborated to create 3 rules corresponding to the 3 elastic pipeline stages.

4.5.3 Folded IFFT

The folded version of IFFT can be written as follows:

Elastic pipelined IFFT

```
module ...
   ...
   Reg #(Bit #(2))                  stage <- mkRegU;
   Reg #(Vector#(64, Complex#(n)))  sReg  <- mkRegU;

   rule foldedEntry (stage==0);
      sReg <= stage_f (stage, inQ.first()); stage <= stage+1;
      inQ.deq();
   endrule

   rule foldedCirculate ((stage!=0) && (stage != (n-1)));
      sReg <= stage_f (stage, sReg);  stage <= stage+1;
   endrule

   rule foldedExit (stage==n-1);
      outQ.enq (stage_f (stage, sReg));  stage <= 0;
   endrule
   ...
endmodule
```

The rule `foldedEntry` takes data from FIFO `inQ` (when available) and performs the first stage of the computation. The rule `foldedCirculate` performs the second stage of the computation, and the rule `foldedExit` performs the final stage of the computation. Note that the three rules have mutually exclusive rule conditions, resulting in sequential execution. In general, for computations that have $n > 3$ stages, rule `foldedCirculate` will execute repeatedly for all except the first and last stages.

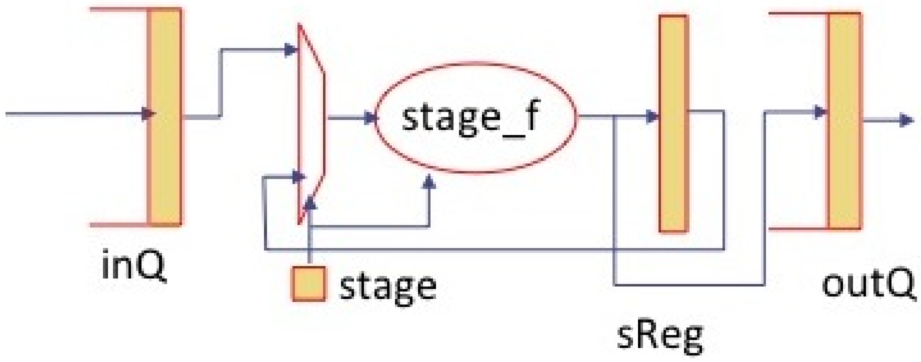


Figure 4.7: Folded IFFT

The resulting folded circuit is illustrated in Fig. 4.7.

4.5.4 Super-folded IFFT

Inside the `stage_f` function in each of the above architectures, there is a column of 16 Bfly4 circuits, as illustrated on the left side of Fig. 4.8.

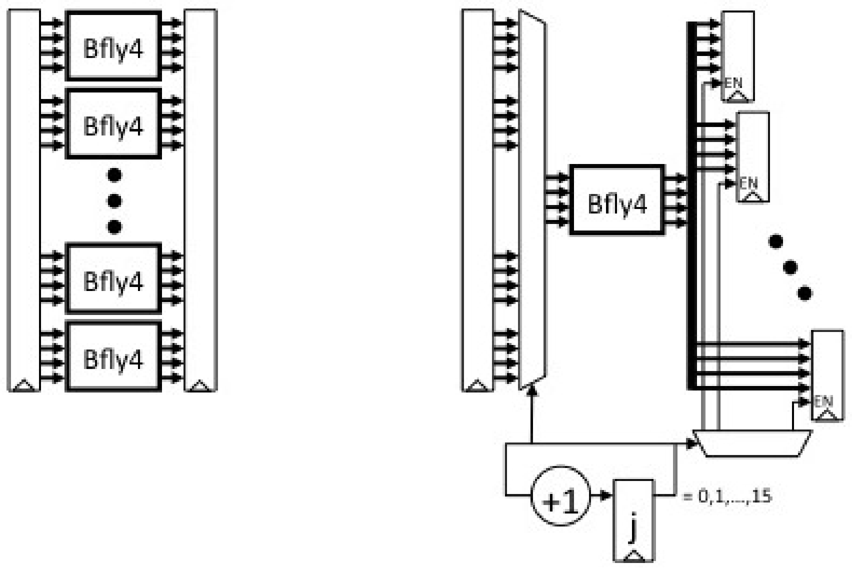


Figure 4.8: Bfly4s in super-folded IFFT

Recall that the input is a vector of 64 complex numbers and that each Bfly4 takes 4 of these and produces an output of 4 complex numbers. The circuit on the right side of Fig. 4.8 shows how we can implement the same function with just one instance of the Bfly4 circuit. The *j* register counts up from 0 to 15. For each *j*, a multiplexer selects a group of 4 complex numbers from the input vector which gets sent into the Bfly4 circuit. The output of the Bfly4 circuit is fed to 16 groups of registers, each of which can hold 4 complex numbers, but it only gets loaded into the j^{th} register because the demultiplexer only enables that register.

The super-folded version decreases area in Bfly4 circuits (one instead of 16 instances), but increases area due to the control logic (multiplexer, demultiplexer, j register, incrementer); whether this is a net savings or an increase depends on the relative sizes of these components. The throughput of the super-folded version is at most 1/16 of the original version, since we are serializing the Bfly4s where we were doing them in parallel.

The following BSV code is a sketch of the super-folded version:

Super-folding the Bfly4s in IFFT

```
module ...
   ...
   Reg #(Vector #(64, Complex#(n))) rg_in <- mkRegU;
   Vector #(16, Reg #(Vector #(4, Complex #(n)))) rg_outs
                                                <- replicateM (mkRegU);
   Reg #(Bit#(5)) j <- mkRegU;

   rule step_Bfly4 (j != 16);
      let x0 = rg_in[j*4];     let x1 = rg_in[j*4+1];
      let x2 = rg_in[j*4+2];   let x3 = rg_in[j*4+3];
      Vector #(4, Complex#(n)) y = bfly4 (x0,x1,x2,x3);
      rg_outs[j] <= y;
      j <= j + 1;
```

```
    endrule
    ...
endmodule
```

Line 4 declares `rg_outs` as a vector of 16 registers, each of which holds a vector of 4 complex numbers. Note that the `Vector` type constructor is used in one case on values (`Complex#(n)`) and in the other case on interfaces (`Reg#(...)`). Line 5 instantiates the 16 registers. When j is initialized to 0 (by some other rule), the rule `step_Bfly4` will then execute 16 times sequentially, performing each of the 16 steps described above.

4.5.5 Comparing all the architectural variants of IFFT

People familiar with software compilers will recognize that architectural unfolding is analogous to the standard compiler techniques of "inlining" and "loop unrolling". In fact, they have the same attendant risks and benefits. Unfolding can increase the footprint (area), but it can also enable certain optimizations such as constant propagation (an variable parameter to a folded circuit may become a constant (perhaps a different constant) in each unfolded instance, which may lead to a decrease in area. In Sec. 4.5.4 we discussed how folding can decrease area (fewer instances of the repeated computation circuit), but also increases area due to the control logic to manage sequential execution. For these reasons, it may be hard to predict which point in the space of possible architectures is "best" for a given application.

By applying pipelining and folding to IFFT circuit as described above, there are 24 different architectural variants, all implementing exactly the same IFFT function (exactly the same arithmetic transformation of inputs to outputs). In [4], the authors show actual silicon area measurements for several variants. The following is an excerpt (the exact area units are unimportant, just the relative sizes). The two columns represent silicon area used in combinational circuits and sequential circuits, respectively.

	Comb. area	Seq. area
Combinational	16,536	7,279
Folded	29,330	11,603
Pipelined	20,610	18,558

Surprisingly, the combinational version (which is fully unfolded) is smaller than the folded version. The authors say that it is due to the phenomenon discussed earlier, where unfolding enables opportunities for aggressive constant-folding optimizations, reducing the size of each Bfly4 by 60

The techniques we have discussed in this chapter—elastic or inelastic pipelining, and folding—are standard "design patterns" in hardware design and are used in all kinds of circuits. You will see more examples of this in the chapters to come.

Of course, a design may contain both pipelined and folded components. Pipelining will be favored for parts that are frequently executed and need high throughput, whereas folding will be favored for parts that may be infrequently executed and/or where unfolded area would be prohibitively expensive.

Advanced topic: pipeline "design patterns"

In modern programming languages with sufficient expressive power, design patterns can be captured as so-called "higher-order functions" or "parameterized functions". Thus, we don't reimplement a design pattern every time we need one. Instead, we just use a library function, passing it parameters that specialize it for our particular use-case.

For example, the "pipeline design pattern" is just a function parameterized by the list of combinational functions to be plugged in to each stage of the pipeline. The "folded for-loop design pattern" is just a function parameterized by the loop-count and the pipeline that should be plugged in as the body of the loop. The "folded while-loop design pattern" is similar, except that, instead of the loop-count, the parameter is a boolean function to test whether the loop computation has completed.

This idea can be expressed in BSV, and exist in a library called *PAClib* (for Pipelined Architecture Constructor Library). We do not discuss this further here, but PAClib is described in a whitepaper from Bluespec, Inc. [3], in which they also show a single, 100-line BSV program for IFFT with 4 static parameters. By suitably choosing these parameters, the program elaborates into one of the 24 possible architectural variants available using the techniques described in this chapter.

Chapter 5

Introduction to SMIPS: a basic implementation without pipelining

5.1 Introduction to SMIPS

We now have enough foundational material to turn to the central focus of this book, namely, processor architectures. An immediate choice is: which instruction set? We can invent a new one, or reuse an existing instruction set. The advantages of the latter include validation and credibility (it has been tested in the the field and in commercial products). More importantly, we can reuse existing tool chains (compilers, linkers, debuggers) and system software (operating systems, device drivers) in order quickly to produce realistic programs to run on our system. Real system software and applications are essential for taking meaningful and credible measurements regarding efficiency and performance of hardware implementations.

In this book, we will be implementing an instruction set architecture (ISA) called SMIPS ("Simple MIPS"), which is a subset of the full MIPS32 ISA. MIPS was one of the first commercial RISC (Reduced Instruction Set Computer) processors and continues to be used in a wide range of commercial products such as Casio PDAs, Sony Playstation and Cisco internet routers (see Appendix A for some more history). MIPS implementations probably span the widest range for any commercial ISA, from simple single-issue in-order pipelines to quad-issue out-of-order superscalar processors.

Our subset, SMIPS, does not include floating point instructions, trap instructions, misaligned load/stores, branch and link instructions, or branch likely instructions. There are three SMIPS variants which are discussed in more detail in Appendix A. SMIPSv1 has only five instructions and it is mainly used as a toy ISA for instructional purposes. SMIPSv2 includes the basic integer, memory, and control instructions. It excludes multiply instructions, divide instructions, byte/halfword loads/stores, and instructions which cause arithmetic overflows. Neither SMIPSv1 noor SMIPSv2 support exceptions, interrupts, or most of the system coprocessor. SMIPSv3 is the full SMIPS ISA and includes all the instructions in our MIPS subset.

Appendix A has a detailed technical reference for SMIPS. We now sketch a high-level overview.

5.1.1 Instruction Set Architectures, Architecturally Visible State, and Implementation State

In computer architecture it is very important to keep a clear distinction between an Instruction Set Architecture (ISA) and its many possible implementations.

An ISA is just a definition or *specification* of an instruction set: What is the repertoire of instructions? What state elements are manipulated by instructions ("architecturallly visibie state")? How are instructions encoded into bits in memory? What is the meaning of the execution of each instruction (*i.e.,* how does it manipulate the architectural state)? What is the meaning of sequential execution of instructions?

Note that an ISA typically says nothing about clocks or pipelining, nor about superscalarity (issueing and executing multiple instructions per clock), nor out-of-order execution (execution in a different order from the semantic ISA specification), branch prediction, load-value prediction, nor any of a thousand other techniques an implementation may use to maximize performance, or efficiency, or both. Each of these implementation techniques will typically introduce more state elements into the processor (pipeline registers, replicated and shadow copies of registers for faster access, and more), but none of these are visible in the ISA, *i.e.,* these state elements are purely internal implementation artefacts. We thus make a distinction between "architecturally visible" state—state that can be named and manipulated in the ISA—versus "implementation state" or "micro-architectural state" which includes all the implementation-motivated state in the processor.

An ISA typically plays the role of a stable "contract" between software developers and processor implementers. Software developers—whether they directly write machine code or assembly code, or whether they write compilers that produce machine code, or whether they write application codes using compilers developed by others—use the ISA as an unambiguous specification of what their machine code programs will do. They do not have to react and revise their codes every time there is a new implementation of the given ISA.

Processor implementers, on the other hand, are free to explore new micro-architectural ideas to improve the performance or efficiency of their next implementation of the ISA.

Another way to think of this is that the archtecturally visible state is what you see when you use a debugger such as gdb. These are precisely the registers described in the assembly language programmer's manual, and memory. You typically cannot see pipeline registers, FIFOs for memory requests and responses, branch prediction state, and so on. Further, when you "single-step" in a debugger such as gdb, the smallest unit is a single, complete, assembly lnaguage instruction; you typically cannot pause after an instruction fetch, or when the instruction is halfway through the processor pipeline.

This is not to say that ISAs never change. ISAs do get extended in response to new market needs. Typical examples are the new opcodes that have found their way into most modern instruction sets for graphics, high-performance computation (SIMD), encryption, and so on. But this change in ISAs is usually a relatively slow, deliberative, evolutionary process (measured in years and decades), so that implementers still have a stable target to implement within typical implementation timeframes.

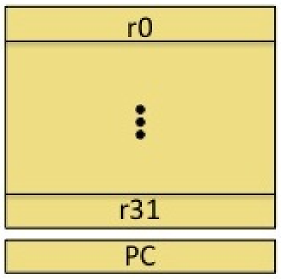


Figure 5.1: Basic SMIPS processor architectural state

5.1.2 SMIPS processor architectural state

Fig. 5.1 shows the basic architecturally visible state in an SMIPS processor. There are 32 general-purpose, 32-bit registers (GPRs), called `r0` through `r31`. When read, `r0` always returns a zero, and so is useful for instructions needing the constant zero. The Program Counter, or PC, is a separate 32-bit register. And, there are a few other special-purpose registers, not illustrated here.

The "native" data types manipulated in the ISA are 8-bit bytes, 16-bit half-words and 32-bit words. All instructions are encoded as 32-bit values. The ISA is a so-called "Load-Store" ISA which is typical of RISC ISAs, namely that the opcodes are paritioned into one set of instructions that only move data between memory and registers (Load/Store), and another set of instructions that only perform ALU or control operations on data in registers. The Load/Store instructions have immediate and indexed addressing modes. For branch instructions, branch targets can be relative to the PC or indirect via an address in a register.

5.1.3 SMIPS processor instruction formats

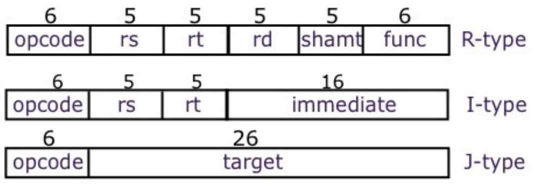


Figure 5.2: SMIPS processor instruction formats

There are are only 3 kinds of instruction encodings in SMIPS, shown in Fig. 5.2. However, the fields are used differently by different types of instructions.

Fig. 5.3 shows computational and load/store instructions. The first form of computational instruction is a classical "3-operand" instruction, with two source registers and one destination register, and the opcode specified in the func field. The second form is a 2-operand instruction where the 3^{rd} operand is an immediate value (taken from bits in the instruction itself). In the load/store instructions, the load/store address is computed by adding rs and the displacement. For loads, rt is the destination register receiving the loaded value. For stores, rt is the source register containing the value to be stored.

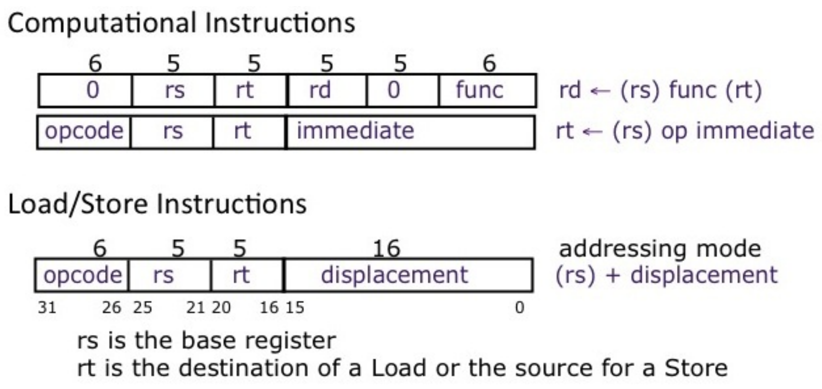

Computational Instructions
6 5 5 5 5 6
0 | rs | rt | rd | 0 | func
rd ← (rs) func (rt)
opcode | rs | rt | immediate
rt ← (rs) op immediate
Load/Store Instructions
6 5 5 16
opcode | rs | rt | displacement
addressing mode
(rs) + displacement
31 26 25 21 20 16 15 0
rs is the base register
rt is the destination of a Load or the source for a Store

Figure 5.3: SMIPS processor Computational and Load/Store instruction formats

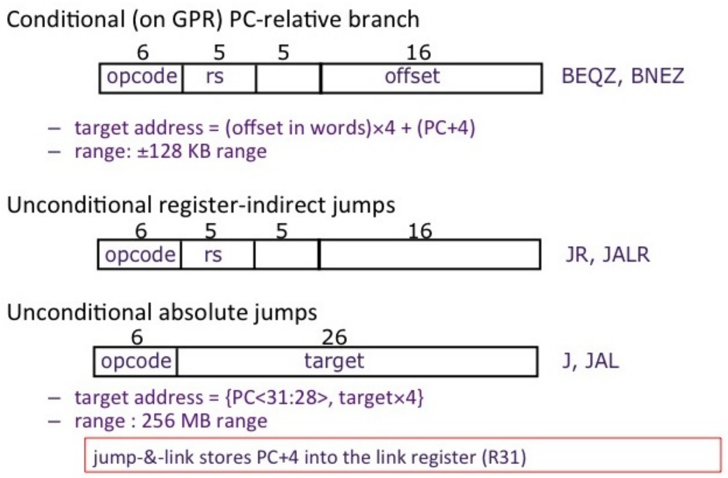

Conditional (on GPR) PC-relative branch
6 5 5 16
opcode | rs | | offset
BEQZ, BNEZ
– target address = (offset in words)×4 + (PC+4)
– range: ±128 KB range
Unconditional register-indirect jumps
6 5 5 16
opcode | rs | |
JR, JALR
Unconditional absolute jumps
6 26
opcode | target
J, JAL
– target address = {PC<31:28>, target×4}
– range : 256 MB range
jump-&-link stores PC+4 into the link register (R31)

Figure 5.4: SMIPS processor control (branch) instruction formats

Fig. 5.4 shows control (branch) instructions. The first form tests register rs and conditionally branches to a target that is an offset from the current PC. The second form is an unconditional jump based on a target address in register rs. One of its uses is for returning from a function call, where rs represents the return address. The third form is an unconditional jump based on an absolute target address. The JAL (Jump And Link) opcode also saves PC+4 into r31 (also known as the Link register); this can be used for function calls, where PC+4 represents the return address.

5.2 Uniform interface for our processor implementations

Over the next few chapters we will be describing a series of increasingly sophisticated computer systems, each containing a processor implementation and instruction and data memories. All of them will have a uniform interface with the test environment, *i.e.,* a uniform way by which:

- a "memory image" (including the machine code for a program) is initially loaded into the computer's memory;
- the testbench "starts" the processor, allowing it to execute the program, and
- the testbench waits for the processor to complete execution of the program (or aborts the program after some time limit)

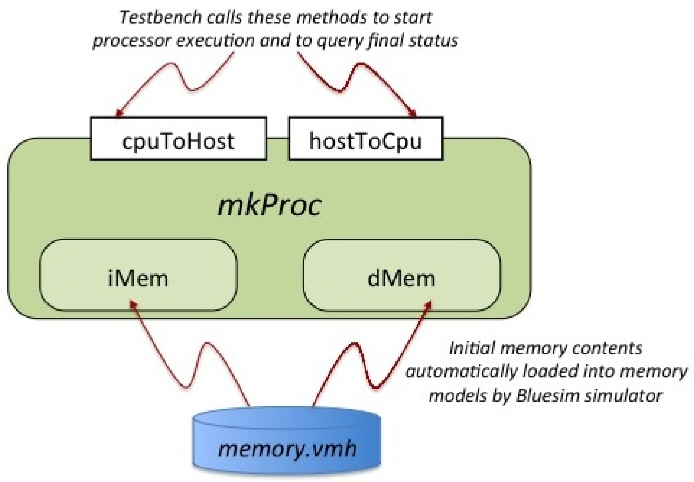

Testbench calls these methods to start processor execution and to query final status
cpuToHost
hostToCpu
mkProc
iMem
dMem
Initial memory contents automatically loaded into memory models by Bluesim simulator
memory.vmh

Figure 5.5: Uniform interface for running our implementations

Fig. 5.5 illustrates the uniform test setup. The "top level" of each system we describe will be a BSV module called `mkProc`. In early versions of the code, for simplicity, we will instantiate models of instruction and data memory inside `mkProc` using the module constructors `mkIMemory` and `mkDMemory`. When simulation begins, these modules automatically load a file called `memory.vmh` which is a text file describing the contents of "virtual memory" as a series of 32-bit numbers expressed in hexadecimal format. The I- and D-memories both load a copy of the same file. The lab exercises provide these files for a number of initial memory loads. These initial-load files are typically created by cross-compilers,[1] *i.e.,* we compile a

[1]The compiler is termed a "cross-compiler" because, although it runs on a workstation that is likely *not* an SMIPS machine, the binary file that it produces contains SMIPS instructions.

C application program into SMIPS binary which is placed in such a file which gets loaded into instruction memory.

The mkProc module presents the following interface to the testbench for starting and stopping the processor, and querying its status:

Processor Interface (in ProcTypes.bsv)

```
interface Proc;
   method Action hostToCpu (Addr startpc);
   method ActionValue #(Tuple2#(RIndx, Data)) cpuToHost;
endinterface
```

The hostToCpu method allows the testbench to start the processor's execution, providing it a starting PC address. The cpuToHost method returns the final status of the processor after it has halted.

5.3 A simple single-cycle implementation of SMIPS v1

We begin our series of increasingly sophisticated processor implementations with the simplest one—where all the functionality is expressed within a single BSV rule. It is useful as a warm-up exercise, of course, but it is also useful as a "reference implementation" against which to compare future, more complex implementations. In other words, it provides a reference specification for what the final architecturally visible state of the processor ought to be after executing a particular program starting with a particular initial state. This one-rule implementation is a BSV counterpart to a traditional "Instruction Set Simulator" (ISS), often written in C/C++, that serves as a simples reference implementation of the ISA semantics (with no complications due to hardware implementation considerations).

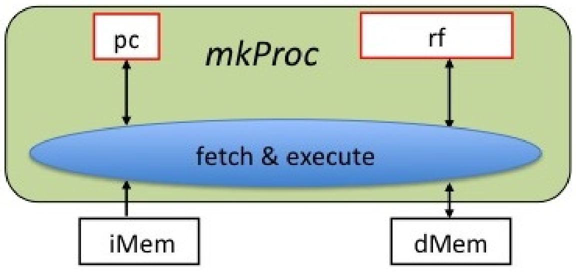


Figure 5.6: SMIPS processor 1-rule implementation

Fig. 5.6 illustrates the structure of our first implementation. It instantiates a PC register, a register file rf, and instruction and data memories iMem and dMem respectively, *i.e.*, nothing more than the ISA-specified architecturally-visible state.

To execute a program, the processor must repeatedly do the following steps:

- Instruction Fetch: read the 32-bit word in Instruction Memory at address PC).
- Instruction Decode: separate it into its constituent fields.

- Register Read: read the source registers mentioned in the instruction, from the register file. Since many instructions have more than one source register, this requires the register file to have more than one port, for simultaneous access.
- Execute: perform any ALU op required by the instruction. This may be specified by an ALU opcode, a comparision opcode, or an address calculation for a load/store, etc.
- Memory operation: if it is a load/store instruction, perform the memory operation.
- Write back: for instructions that have destination registers, write the output value to the specified register in the register file.
- Update the PC: increment the PC (normally) or replace it with a new target value (branch instructions).

Why do we separate instruction and data memories into two separate memories? Although in principle all modern processors have the functionality of Turing Machines in that conceptually there is just a single memory and they can read and write any location in memory, in modern practice we rarely do this. Specifically, if we were to write into the part of memory holding instructions ("self-modifying code"), it would greatly complicate implementation because it would add a potential interaction between the "Instruction Fetch" and "Memory Operation" stages in the list above (the next instruction could not be fetched until the memory operation stage makes its modification). Modern practice avoids self-modifying code completely, and the part of memory that holds instructions is often protected (using virtual memory protection mechanisms) to prevent writes. In this view, therefore, instruction and data accesses can be performed independently, and hence we model them as separate memories. For reasons going back to influential early research computers developed at Harvard and Princeton Universities in the 1940s, separated-memory and unified-memory architectures are called Harvard and Princeton architectures, respectively. In this book we shall following modern practice and only discus Harvard architectures.

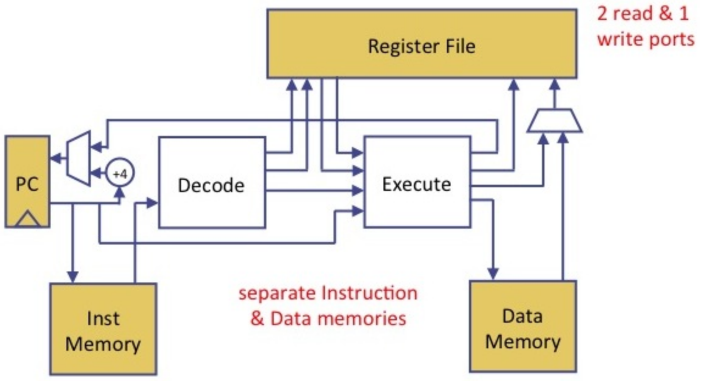


Figure 5.7: SMIPS processor 1-rule implementation datapaths

Fig. 5.7 illustrates the datapaths that will emerge for our 1-rule implementation. Of course, these datapaths are automatically created by the *bsc* compiler when compiling our BSV code. Here is the BSV code for our 1-rule implementation:

1 cycle implementation (in 1cyc.bsv)
```
module mkProc(Proc);
  Reg#(Addr) pc <- mkRegU;
  RFile      rf <- mkRFile;
  IMemory  iMem <- mkIMemory;
  DMemory  dMem <- mkDMemory;
  Cop       cop <- mkCop;

  rule doProc(cop.started);
    let inst = iMem.req(pc);

    // decode
    let dInst = decode(inst);

    // trace - print the instruction
    $display("pc: %h inst: (%h) expanded: ", pc, inst, showInst(inst));

    // read register values
    let rVal1 = rf.rd1(validRegValue(dInst.src1));
    let rVal2 = rf.rd2(validRegValue(dInst.src2));

    // Co-processor read for debugging
    let copVal = cop.rd(validRegValue(dInst.src1));

    // execute
    // The fifth argument is the predicted pc, to detect if it was
    // mispredicted (future). Since there is no branch prediction yet,
    // this field is sent with an unspecified value.
    let eInst = exec(dInst, rVal1, rVal2, pc, ?, copVal);

    // Executing unsupported instruction. Exiting
    if(eInst.iType == Unsupported)
    begin
      $fwrite(stderr, "Unsupported instruction at pc: %x. Exiting\n", pc);
      $finish;
    end

    // memory
    if(eInst.iType == Ld) begin
      eInst.data <- dMem.req(MemReq{op: Ld, addr: eInst.addr, data: ?});
    end
    else if(eInst.iType == St) begin
      let d <- dMem.req(MemReq{op: St, addr: eInst.addr, data: eInst.data});
    end

    // write back
    if(isValid(eInst.dst) && validValue(eInst.dst).regType == Normal)
      rf.wr(validRegValue(eInst.dst), eInst.data);

    // update the pc depending on whether the branch is taken or not
    pc <= eInst.brTaken ? eInst.addr : pc + 4;
```

```
51
52      // Co-processor write for debugging and stats
53      cop.wr(eInst.dst, eInst.data);
54    endrule
55
56    method ActionValue#(Tuple2#(RIndx, Data)) cpuToHost;
57      let ret <- cop.cpuToHost;
58      $display("sending %d %d", tpl_1(ret), tpl_2(ret));
59      return ret;
60    endmethod
61
62    method Action hostToCpu(Bit#(32) startpc) if (!cop.started);
63      cop.start;
64      pc <= startpc;
65    endmethod
66  endmodule
```

Line 1 declares the `mkProc` module with the previously described `Proc` interface. Lines 2-6 instantiate the sub-modules: the PC register, a register file, the instruction and data memories, and `cop`, a standard *co-processor*. The co-processor is the administrative controller of the processor, allowing it to start and stop, and keeps a record of time (cycle count), number of instructions executed, and so on.

Lines 8-54 specify a single rule that does everything, and it is only enabled when `cop.started` is true. It more-or-less follows the itemized list described earlier: Line 9 is the Instruction Fetch; line 12 is Instruction Decode. Lines 18-19 perform the two source register reads (the `validRegValue` functions are needed because not all instructions have two source registers). Ignore line 22 for now; it's using the co-processor for debugging. Line 28 is Execute. Lines 30-35 detect bad instructions (not supported in this ISA). Lines 37-43 do the memory operation, if any. Line 46-47 do the Write Back, if any, and line 50 updates the PC. Lines 56-65 implement the `Proc` interface methods.

Note that all the functions and methods used in this rule including `iMem.req()`, `decode()`, `rf.rd1()`, `rf.rd2()`, `exec()`, `dMem.req()`, and `rf.wr()` are instantaneous (purely combinational). In fact they must be, since they are in a single rule.

Let us dig a little deeper into the BSV code. We start with some type definitions:

Some type definitions (in ProcTypes.bsv)

```
typedef Bit#(5) RIndx;

typedef enum {Unsupported, Alu, Ld, St, J, Jr, Br, Mfc0, Mtc0} IType
        deriving(Bits, Eq);
typedef enum {Eq, Neq, Le, Lt, Ge, Gt, AT, NT} BrFunc
        deriving(Bits, Eq);
typedef enum {Add, Sub, And, Or, Xor, Nor,
              Slt, Sltu, LShift, RShift, Sra} AluFunc
        deriving(Bits, Eq);
```

Line 1 defines a register index to be a 5-bit value (since we have 32 registers in SMIPS). The next few lines describe symbolic names for various opcode functions. The following code describes a decoded instruction:

Decoded instructions (in ProcTypes.bsv)

```
typedef struct {
  IType            iType;
  AluFunc          aluFunc;
  BrFunc           brFunc;
  Maybe#(FullIndx) dst;
  Maybe#(FullIndx) src1;
  Maybe#(FullIndx) src2;
  Maybe#(Data)     imm;
} DecodedInst deriving(Bits, Eq);
```

The `decode()` function is in the file `Decode.bsv` and has the following outline:

Decode function outline (in Decode.bsv)

```
function DecodedInst decode(Data inst);
  DecodedInst dInst = ?;
  let opcode = inst[ 31 : 26 ];
  let rs     = inst[ 25 : 21 ];
  let rt     = inst[ 20 : 16 ];
  let rd     = inst[ 15 : 11 ];
  let shamt  = inst[ 10 :  6 ];
  let funct  = inst[  5 :  0 ];
  let imm    = inst[ 15 :  0 ];
  let target = inst[ 25 :  0 ];

  case (opcode)
     ... fill out the dInst structure accordingly ...
  endcase
  return dInst;
endfunction
```

We show below one of the `case` arms, corresponding to I-Type ALU instructions:

Decode function; I-type section (in Decode.bsv)

```
  ...
  case (opcode)
    opADDIU, opSLTI, opSLTIU, opANDI, opORI, opXORI, opLUI:
    begin
      dInst.iType = Alu;
      dInst.aluFunc = case (opcode)
        opADDIU, opLUI: Add;
        opSLTI: Slt;
        opSLTIU: Sltu;
        opANDI: And;
        opORI: Or;
        opXORI: Xor;
      endcase;
      dInst.dst  = validReg(rt);
      dInst.src1 = validReg(rs);
      dInst.src2 = Invalid;
      dInst.imm = Valid(case (opcode)
        opADDIU, opSLTI, opSLTIU: signExtend(imm);
        opLUI: {imm, 16'b0};
        default: zeroExtend(imm);
      endcase);
      dInst.brFunc = NT;
    end
    ... other case arms ...
  endcase
```

Please refer to the file `Decode.bsv` to see the full function. At the bottom of the function is a case-arm "catch all" that handles bad instructions, *i.e.*, 32-bit encodings that do not conform to any of the specified instructions:

`Decoding unsupported instructions (in Decode.bsv)`

```
1    ...
2    case
3      ... other case arms
4
5      default:
6      begin
7        dInst.iType = Unsupported;
8        dInst.dst  = Invalid;
9        dInst.src1 = Invalid;
10       dInst.src2 = Invalid;
11       dInst.imm  = Invalid;
12       dInst.brFunc = NT;
13     end
14   endcase
15
16   if (dInst.dst matches tagged Valid .dst
17       &&& dst.regType == Normal
18       &&& dst.idx == 0)
19     dInst.dst = tagged Invalid;
20
21   return dInst;
22 endmodule
```

In lines 16-19 we handle the case where the destination register has been specified as `r0` in the instruction; we convert this into an "invalid" destination (which will cause the write back to ignore this).

Fig. 5.8 illustrates the `exec()` function (purely combinational). The output of the function is a structure like this:

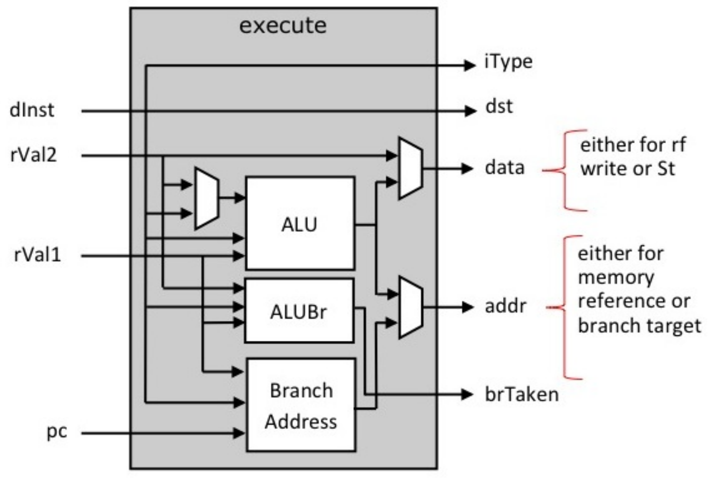

execute
dInst
rVal2
rVal1
pc
ALU
ALUBr
Branch Address
iType
dst
data
addr
brTaken
either for rf write or St
either for memory reference or branch target

Figure 5.8: SMIPS processor Execute function

```
Executed Instructions (in ProcTypes.bsv)
typedef struct {
  IType             iType;
  Maybe#(FullIndx) dst;
  Data              data;
  Addr              addr;
  Bool              mispredict;
  Bool              brTaken;
} ExecInst deriving(Bits, Eq);
```

The code below shows the top-level of the `exec` function:

```
Execute Function (in Exec.bsv)
function ExecInst exec(DecodedInst dInst, Data rVal1, Data rVal2, Addr pc,
                       Addr ppc, Data copVal);
  ExecInst eInst = ?;
  Data aluVal2 = isValid(dInst.imm) ? validValue(dInst.imm) : rVal2;

  let aluRes = alu(rVal1, aluVal2, dInst.aluFunc);

  eInst.iType = dInst.iType;

  eInst.data = dInst.iType == Mfc0?
                 copVal :
               dInst.iType == Mtc0?
                 rVal1 :
               dInst.iType==St?
                 rVal2 :
               (dInst.iType==J || dInst.iType==Jr) ?
                 (pc+4) :
                 aluRes;
```

```
19
20    let brTaken = aluBr(rVal1, rVal2, dInst.brFunc);
21    let brAddr = brAddrCalc(pc, rVal1, dInst.iType,
22                             validValue(dInst.imm), brTaken);
23    eInst.mispredict = brAddr != ppc;
24
25    eInst.brTaken = brTaken;
26    eInst.addr = (dInst.iType == Ld || dInst.iType == St) ? aluRes : brAddr;
27
28    eInst.dst = dInst.dst;
29
30    return eInst;
31  endfunction
```

In lines 4-6 we use the `alu()` function to compute an ALU result, assuming it is an ALU-type instruction. In lines 10-18, we set the output `eInst.data` field, which could be from the co-processor, `rVal1` or `rVal2`, a branch target (PC+4) for J and Jr instructions, or the ALU output `aluRes`. In lines 20-25, we compute the branch operation, assuming it's a branch instruction. Finally we set `eInst.addr` in line 26, which is the `aluRes` for Ld and St instruction, else the branch address.

For readers with a purely software background, the above code may appear a little strange in that we seem to be performing both the ALU and the branch computations, whatever the opcode. But remember that this is how hardware is typically structured: both computations always exists (they are both statically elaborated and laid out in hardware) and they both do their work, and the final result is just selected with a multiplexer.

The `alu()` and branch functions used above are shown next:

ALU and Branch parts of Execute Function (in Exec.bsv)

```
function Data alu(Data a, Data b, AluFunc func);
  Data res = case(func)
     Add   : (a + b);
     Sub   : (a - b);
     And   : (a & b);
     Or    : (a | b);
     Xor   : (a ^ b);
     Nor   : ~(a | b);
     Slt   : zeroExtend( pack( signedLT(a, b) ) );
     Sltu  : zeroExtend( pack( a < b ) );
     LShift: (a << b[4:0]);
     RShift: (a >> b[4:0]);
     Sra   : signedShiftRight(a, b[4:0]);
  endcase;
  return res;
endfunction

function Bool aluBr(Data a, Data b, BrFunc brFunc);
  Bool brTaken = case(brFunc)
```

```
    Eq  : (a == b);
    Neq : (a != b);
    Le  : signedLE(a, 0);
    Lt  : signedLT(a, 0);
    Ge  : signedGE(a, 0);
    Gt  : signedGT(a, 0);
    AT  : True;
    NT  : False;
  endcase;
  return brTaken;
endfunction

function Addr brAddrCalc(Addr pc, Data val, IType iType, Data imm, Bool taken);
  Addr pcPlus4 = pc + 4;
  Addr targetAddr = case (iType)
    J  : {pcPlus4[31:28], imm[27:0]};
    Jr : val;
    Br : (taken? pcPlus4 + imm : pcPlus4);
    Alu, Ld, St, Mfc0, Mtc0, Unsupported: pcPlus4;
  endcase;
  return targetAddr;
endfunction
```

5.4 Expressing our single-cycle CPU with BSV, versus prior methodologies

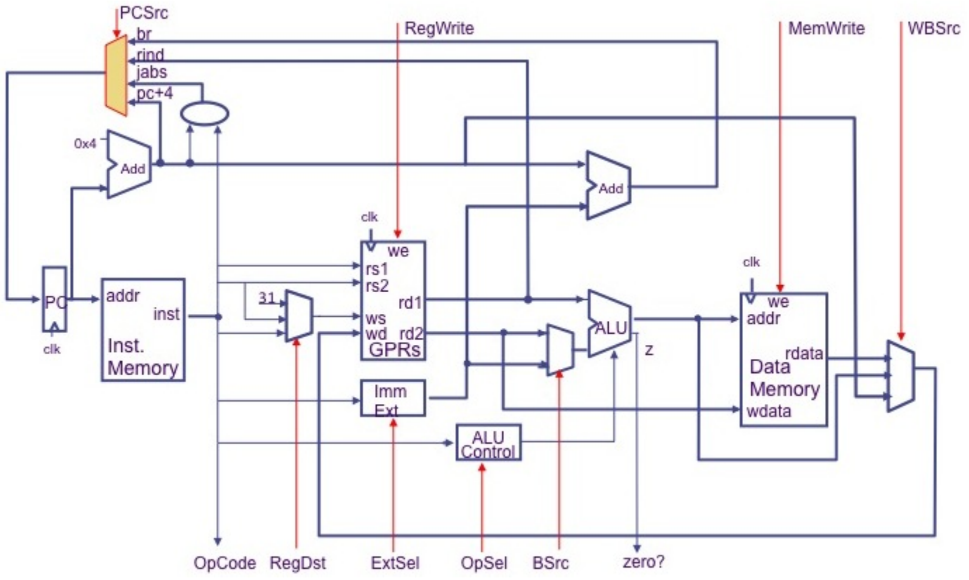
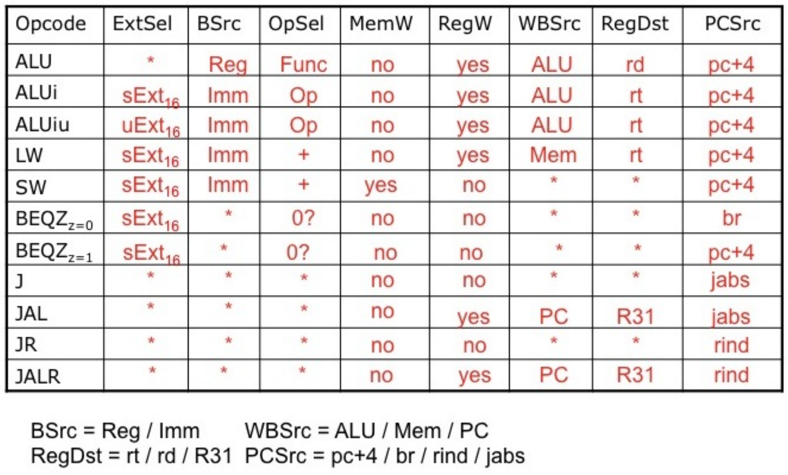


Figure 5.9: Datapath schematic for 1-cycle implementation

In the old days of CPU design, a designer may have begun the process by first drawing a schematic of the datapath of the single-cycle CPU, as illustrated in Fig. 5.9. The red vertical arrows represent control signals—by asserting/deasserting them appropriately, one can steer data from source state elements into chosen functions and into chosen destination state elements.



Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSrc
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	sExt_{16}	Imm	Op	no	yes	ALU	rt	pc+4
ALUiu	uExt_{16}	Imm	Op	no	yes	ALU	rt	pc+4
LW	sExt_{16}	Imm	+	no	yes	Mem	rt	pc+4
SW	sExt_{16}	Imm	+	yes	no	*	*	pc+4
$\text{BEQZ}_{z=0}$	sExt_{16}	*	0?	no	no	*	*	br
$\text{BEQZ}_{z=1}$	sExt_{16}	*	0?	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	R31	jabs
JR	*	*	*	no	no	*	*	rind
JALR	*	*	*	no	yes	PC	R31	rind

BSrc = Reg / Imm WBSrc = ALU / Mem / PC
RegDst = rt / rd / R31 PCSrc = pc+4 / br / rind / jabs

Figure 5.10: Control table for Datapath Schematic for 1-cycle implementation

Next, the designer may create a "control table" as shown in Fig. 5.10, showing what values to set the control signals under each possible set of inputs coming from the circuit (signals like OpCode, zero?, etc.). This control table can be implemented, for example, in a ROM, with the inputs representing the address and the outputs representing the control signals (or, the ROM could be further optimized into custom logic).

You can imagine the sheer labor involved in creating such a datapath schematic and control table. And you can imagine the labor involved in accommodating change: to fix a bug in the logic, or to add new functionality.

In the 1990s people started changing this methodology to describe these circuits in RTL languages instead (Verilog and VHDL). Although this greatly reduced the labor of creating and maintaining schematics and control tables, RTL is not much of a behavioral abstraction above schematics in that it has the same clock-based, globally synchronous view of the world. Similar timing errors and race conditions can occur almost as easily with RTL as with schematics.

The description of these circuits in BSV is a radical shift in how we think about and specify circuits. In addition to powerful types, object-oriented module structure and static elaboration, BSV's rules are a dramatically different way of thinking of circuit behavior in terms of instantaneous atomic transactions. Datapaths and control logic are automatically synthesized from such specifications by *bsc*.

5.5 Separating the Fetch and Execute actions

Let us turn our attention back to our BSV description, and think about the performance of our 1-rule design. Essentially the steps itemized in the list in Sec. 5.3 (Instruction Fetch, Decode, Register Read, Execute, Mem, Write Back) form a long sequential chain of logic, since each step depends on the previous one. Thus, the total combinational delay represents a limit on how fast we can clock the state elements; we must wait for the full combinational delay to allow signals to propagate all the way through:

$$t_{clock} > t_{fetch} + t_{dec} + t_{regread} + t_{exec} + t_{mem} + t_{wb}$$

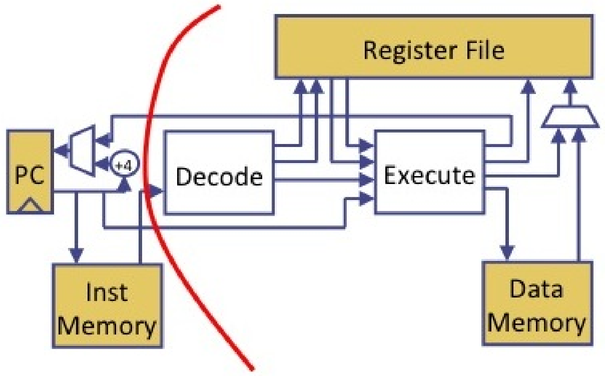


Figure 5.11: Splitting our implementation into 2 rules

Suppose we split our rule into two parts, as illustrated in Fig. 5.11, *i.e.,* separate the Fetch part into its own rule. Then our timing equation changes to:

$$t_{clock} > max(t_{fetch}, t_{dec} + t_{regread} + t_{exec} + t_{mem} + t_{wb})$$

i.e., it would allow us potentially to run the overall circuit at a higher clock speed.

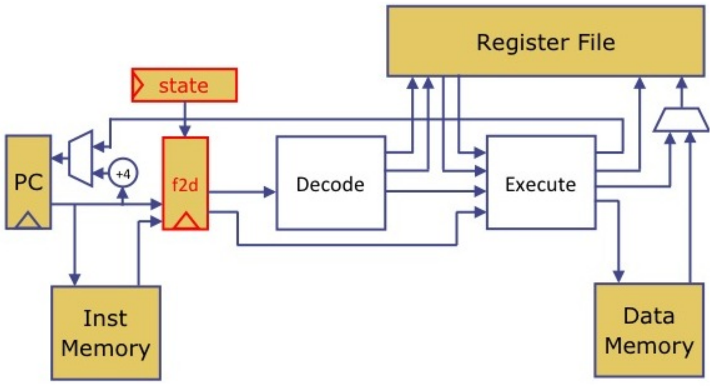


Figure 5.12: Extra state for splitting our implementation into 2 rules

In order to accomplish this, we introduce a register `f2d` to hold the output of the Fetch stage (the 32-bit value from iMem), as shown in Fig. 5.12. We also introduce another register `state` to keep track of whether we are executing the Fetch stage or one of the later stages. These two, `f2d` and `state`, are our first examples of non-architectural state, i.e., state that is not mentioned in the ISA but is introduced for implementation purposes. Below we show the new version of `mkProc`, which is minimally changed from the original 1-rule version:

2-rule processor (in 2cyc_harvard.bsv)

```
 1  typedef enum {Fetch, Execute} State deriving (Bits, Eq);
 2
 3  module mkProc(Proc);
 4    ... same sub-modules as before ...
 5
 6    Reg#(State) state <- mkReg(Fetch);    // NEW
 7    Reg#(Data)    f2d <- mkRegU;          // NEW
 8
 9    rule doFetch(cop.started && state == Fetch);
10      let inst = iMem.req(pc);
11
12      $display("pc: %h inst: (%h) expanded: ", pc, inst, showInst(inst));
13
14      // store the instruction in a register
15      f2d <= inst;
16
17      // switch to execute state
18      state <= Execute;
19    endrule
20
21    rule doExecute(cop.started && state == Execute);
22      let inst = f2d;
23
24      let dInst = decode(inst);
25      ... rest of this rule is same as original doProc rule in 1-cycle version ...
26      ...
27
28      // switch back to fetch
29      state <= Fetch;
30    endrule
31
32    ...
33  endmodule
```

Line 1 declares a type for the state of the 2-rule system: it is either in the Fetch rule, or it is in the Execute rule. Inside the module, lines 6-7 instantiate the two new registers `state` and `f2d`. The original `doProc` rule is now split into two rules, `doFetch` and `doExecute`; we add `state==Fetch` and `state==Execute` to the rule conditions. At the end of `doFetch`, we store the retrieved instruction in the `f2d` register and change `state` to `Execute`. At the start of the `doExecute` rule we retrieve the just-stored instruction from `f2d`, and proceed as before. At the end of the rule we set `state` to `Fetch` to re-enable the `doFetch` rule.

Thus, the `doFetch` and `doExecute` rules will alternate, performing the fetch and execute functions.

5.5.1 Analysis

Have we really gained anything by splitting into two rules and potentially running them at a faster clock speed? In fact we have not: since the two rules alternate, the total time to execute an instruction becomes two clock cycles, and unless we were able to more than double the clock speed (unlikely!) we will be running no faster than before. And, we're now using more state (and therefore likely more hardware area and power).

However, this exercise has demonstrated one key BSV principle: *refinement.* In BSV, one often refines a design by splitting rules into smaller rules (typically with a goal of increasing clock speed). The strength of rule semantics is that this can often be done independently in different parts of a design, *i.e.,* refining a rule typically does not break other parts of the design because we typically do not have tight timing assumptions (as is common in RTL languages). This allows us incrementally to refine a design towards meeting particular performance goals.

This exercise has also set the stage for the real payoff which we shall start exploring in the next chapter, when we start pipelining the two (and more) rules, so that, while the Execute rule is doing its thing, the Fetch rule is busy fetching the next instruction concurrently within the same clock.

Chapter 6

SMIPS: Pipelined

We ended the last chapter with a 2-rule version of the basic MIPS processor, illustrated in Fig. 5.12. Other than demonstrating how to refine a BSV program by breaking larger rules into smaller rules, it did not buy us anything either in terms of overall performance or area. Performance was not improved since the two rules alternate, based on the `state` register: the total time per instruction is the same, it just happens in two rules one after the other, instead of in one rule. And area likely increased due to the introduction of the non-architectural state registers `state` and `f2d`.

In this chapter we will explore how to execute the two rules concurrently (in the same clock), so that they operate as a pipeline. This should substantially improve the performance (throughput) compared to the original 1-rule version, since the smaller rules should allow us to increase clock speed, and we should be able to complete one instruction on every clock.

6.1 Hazards

When creating a processor pipeline, we typically encounter three kinds of *hazards*:

- *Control Hazards*: We do not really know the next instruction to fetch until we have at least decoded the current instruction. If the decoded instruction is not a branch instruction, the next instruction is likely to be at PC+4. If it is a branch instruction, we may not know the new PC until the end of the Execute stage. Even if it is not a branch instruction, it may raise an trap/exception later in the pipeline (such as divide-by-zero, or page fault), requiring the next instruction to be from a trap/exception handler.
- *Structural Hazard*: Two instructions in the pipeline may require the same resource at the same time. For example, in Princeton architectures, both the Fetch and the Execute stages need to access the same memory.
- *Data Hazard*: For example, if one instruction writes into register 13, a later instruction that reads register 13 must wait until the data has been computed and is available. In general, when different stages of the pipeline read and write common state (the register file being just one example), we must be careful about the order of reads and writes.

These are general issues encountered in many kinds of pipelines, not just processors. Simple, pure "feed-forward" pipelines such as the IFFT pipeline of Chapter 4, or an arithmetic multiplier pipeline, typically do not encounter any of these hazards, because there is no dependence between successive samples flowing through the pipeline. However it is in the very nature of instructions in an ISA that they depend on each other, and so we encounter all these kinds of hazards in their full glory. This leads to significantly more complex mechanisms in order to maintain pipeline flow. In fact, dealing with these issues is the essence of creative computer architecture design.

We will focus on control hazards in this chapter, and address the other types of hazards in later chapters.

6.1.1 Modern processors are distributed systems

Historically, the Computer Science topic of *Distributed Systems* grew in the context of Local Area Networks and Wide Area Networks based on the recognition that, in such systems, communication is not free. In particular,

1. There is no instantly visible or updatable global state. Entities only have local state, and they can only send and receive messages to other entities to query or update remote state.
2. Messages cannot be delivered instantaneously; there is a discernable latency, and this latency may be variable and unpredictable.
3. Even between the same two entities, messages may get reordered, *i.e.,* they may arrive in an order different from the order in which they were sent.
4. Messages may get dropped, spuriously generated, duplicated, or corrupted along the way.

At first glance it may seem startling to refer to a single chip that occupies a few square millimeters of silicon (a modern processor or System-on-a-chip) as a "distributed system", but remember that space and time are relative. Modern processors run at multi-gigahertz speeds, and communicating across a chip has a discernable delay, and is best addressed by message-passing between relatively independent entities instead of via shared, globally visible state. Modern chip designers also talk about "GALS" methodology (Globally Asynchronous, Locally Synchronous), and this is just another manifestation of the same observation. Modern system-level communication protocols, such as PCI Express and Intel's Quick Path Interconnect (QPI) even deal with the complexities of unreliable message delivery (bullet items 3 and 4 above).

In summary, it is fruitful to apply the insights and solutions developed in the field of Distributed Systems to the design of modern processor architectures. This insight pervades our approach. In this chapter, the evolution from a globally updated PC to an "epoch" based solution follows this trajectory.

6.2 Two-stage pipelined SMIPS (inelastic)

Fig. 6.1 illustrates a two-stage pipelined version of our SMIPS processor. Let us compare this with Fig. 5.12. First, we have replaced the "PC+4" function in the Fetch stage with

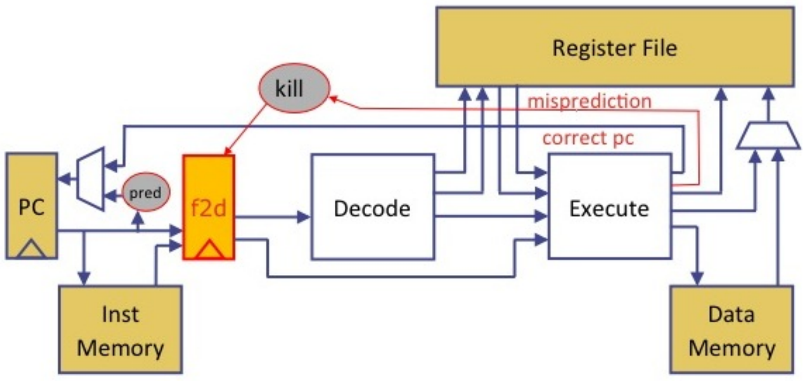


Figure 6.1: Two-stage pipelined SMIPS

a “pred” (for “prediction”) function. The Fetch stage is *speculating* or *predicting* what the next PC will be, and PC+4 is a good guess, since the majority of instructions are not likely to be branch instructions affecting the control flow. Later, we will make more sophisticated predictors that use the current PC to predict the next PC, learning from past instruction executions. For example, a particular branch instruction (at a particular PC) may be executed multiple times (due to loops or repeated calls to the same function), and past behavior may be a good predictor of the next behavior.

But what if we predict wrong, e.g., we predict (in the Fetch stage) that a conditional branch is taken, but we later discover (in the Execute stage) that it is not taken. Then we would have fetched one or more “wrong path” instructions and injected them into the pipeline. We need machinery to kill these instructions (in particular, they should have no effect on any architecturally visible state), and to redirect the Fetch stage to start fetching from the correct PC. In Fig. 6.1 this is depicted as a “kill” function that clears the intermediate `f2d` register, which is the only thing in this design that can hold wrong-path instructions.

Here is some BSV code to implement these ideas; all actions are in a single rule.

1-rule pipelined implementation

```
rule doPipeline;
   // ---- Fetch
   let inst = iMem.req(pc);
   let new_pc = nextAddr(pc);    // predicted PC
   let new_f2d= tagged Valid (Fetch2Decode{pc:pc,ppc:new_pc,inst:inst});

   // ---- Execute
   if (f2d matches tagged Valid .x) begin
      let dInst = decode(x.inst);
      ... register fetch ...;
      let eInst = exec(dInst, rVal1, rVal2, x.pc, x.ppc);
      ...memory operation ...
      ...register file update ...
      if (eInst.mispredict)  begin
         new_f2d = tagged Invalid;
         new_pc = eInst.addr;
```

```
17          end
18       end
19       pc <= new_pc; f2d <= new_f2d;
20    endrule
```

Prediction is encapsulated into the `nextAddr(pc)` function (line 4), which can be as simple as `pc+4`. Redirection happens because we initially bind `new_pc` to the predicted value (line 4), but if we encounter a misprediction, *i.e.,* (`eInst.mispredict` is True (line 14), we rebind `new_pc` to the corrected PC `eInst.addr` (line 16), and we update `pc` to final binding of `new_pc` (line 19). Killing a wrong-path instruction happens because we initially bind `new_f2d` to the decoded (predicted) instruction (line 5), and if we encounter a misprediction we rebind `new_f2d` to `tagged Invalid` (line 15), and this final binding is stored in the `f2d` register (line 19). Pipelining happens because Fetch and Execute can both execute in each rule execution.

This solution works, but is quite complex in lumping all the logic into one big rule. The design methodology does not scale—it will be very difficult to extend this to handle multiple pipeline stages, multi-cycle memory accesses and multi-cycle functional units, variable latencies due to cache misses and data-dependencies in functional units, multiple branch predictors and resolvers, etc. The single rule becomes a larger and larger hairy ball of wax, and every change involves reasoning again about the whole design (note that this "globally synchronous" view is indeed the natural view when designing in RTL languages like Verilog and VHDL). It would be simpler and more modular to be able to design the Fetch and Execute stages independently and then to connect them together; we pursue that approach in the next section and the rest of this book.

6.3 Two-stage pipelined SMIPS (elastic)

Recalling our earlier recommendation that modern silicon architectures should be designed as distributed systems, we should really view our two pipeline stages as concurrent, decoupled structures, as shown in Fig. 6.2. The Fetch and Execute stages are independent entities

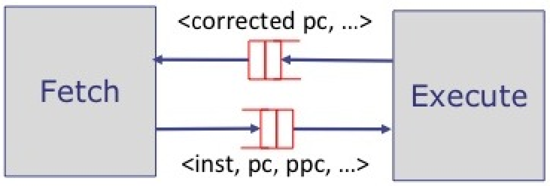


Figure 6.2: Decoupled structure

that communicate using messages. The Fetch unit sends messages to the Execute unit containing (at least) instructions to be executed and information about its PC predictions. The Execute unit, in turn, sends messages back to the Fetch unit containing information about mispredictions and their corrections. The FIFOs connecting the units represent communication channels and, in particular, we make no assumptions about how long they take to deliver the messages.

This last point (unknown message latency) raises a synchronization problem. By the time the backward channel delivers PC-correction information to the Fetch unit, this unit may have already fetched and sent a number of mispredicted instructions into the forward channel and, in fact, some of them may already have been processed in the Execute unit. None of those instructions should have any effect on the architectural state, *i.e.,* they should all effectively be discarded.

There is a subtlety about concurrency that we will gloss over for now. For truly pipelined behavior, the Fetch unit should be able to enqueue a new message into the forward FIFO "concurrently" with the Execute unit dequeueing an older message from the same FIFO. Concurrency of enqueueing and dequeuing on the backward FIFO would also be good, but is perhaps less important because it happens less often (only on mispredictions). For this chapter, we will just assume that these FIFOs are so called "conflict-free" FIFOs, which permit this concurrency. We will discuss concurrency issues in more detail in Chapter 7.

6.3.1 Epochs and epoch registers

There is a standard, simple solution to this problem which is to tag instructions with some information so that the Execute unit knows whether they are mispredicted (and therefore to be discarded) or not. The solution is illustrated in Fig. 6.3. We add an "epoch" register

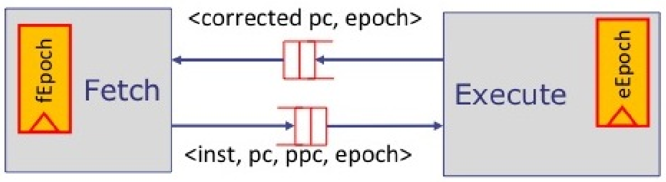


Figure 6.3: Adding epoch registers to distributed units

to each of the distributed units. Think of the "epoch" as counting mispredictions—every time we mispredict a branch, we increment the epoch. Initially, both registers are set to 0.

The Fetch unit tags each instruction sent in the forward channel with the corresponding value of `fEpoch` when the instruction was fetched. When the Execute unit detects a misprediction, it increments `eEpoch` immediately, and sends this new epoch value in the backward channel along with the corrected PC. Note that instructions that are already in the forward channel now have an epoch number that is 1 less than `eEpoch`. The Execute unit discards all such instructions (incoming epoch $<$ `eEpoch`).

When a backward message eventually arrives at the Fetch unit, it updates PC and `fEpoch`, starts fetching from the new PC, and sends messages tagged with this new epoch number (which now matches `eEpoch`). When the Execute unit sees these messages (incoming epoch $=$ `eEpoch`), it stops discarding instructions and resumes normal execution. Thus, mispredicted instructions are correctly discarded, and execution resumes correctly on the correct-path instructions.

Fig.6.4 shows how the values in the two epoch registers evolve as we fetch instructions and send them forward.

The `eEpoch` value is shown in red, and the `fEpoch` value is shown in green. Note that the latter always lags behind the former, ie `fEpoch` $\leq$ `eEpoch`. This is what guarantees safe

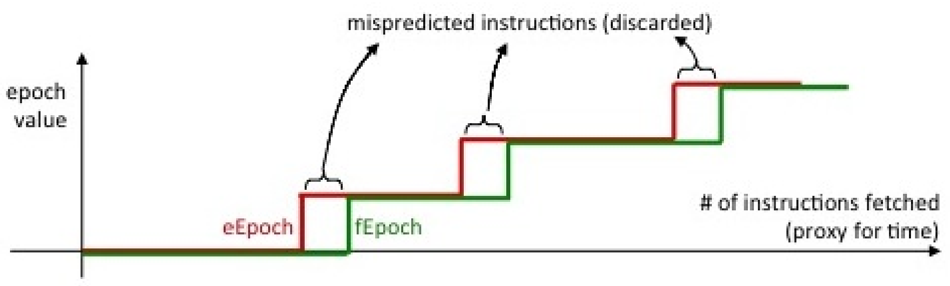


Figure 6.4: Evolution of epoch register values

behavior. In later chapters, as we refine to more pipeline stages and have more sophisticated speculation, each stage will have its own epoch register, and this "$\leq$" property will continue to hold, *i.e.*, each stage's local epoch register will approximate ($\leq$) the value in downstream epoch registers. The last epoch register (here, `eEpoch`) is the "truth", and all upstream epoch registers are safe approximations.

We have described the epoch as a continuously incrementing number. So, how large should we size the `epoch` register (how many bits)? Remember that processors may execute trillions of instructions between reboots. Fortunately, a little thought should persuade us that, for the above 2-stage pipeline, just 1 bit is enough. Across the whole system, at any given time, we are only dealing with two epochs: the latest epoch, and possibly one previous epoch if there are still wrong-path instructions in the pipeline. With a 1-bit representation, updating the epoch register is a just a matter of inverting 1 bit.

6.3.2 Elastic pipeline: two-rules, fully-decoupled, distributed

Fig. 6.5 shows the decoupled architecture using epoch registers. The FIFO `f2d` connects

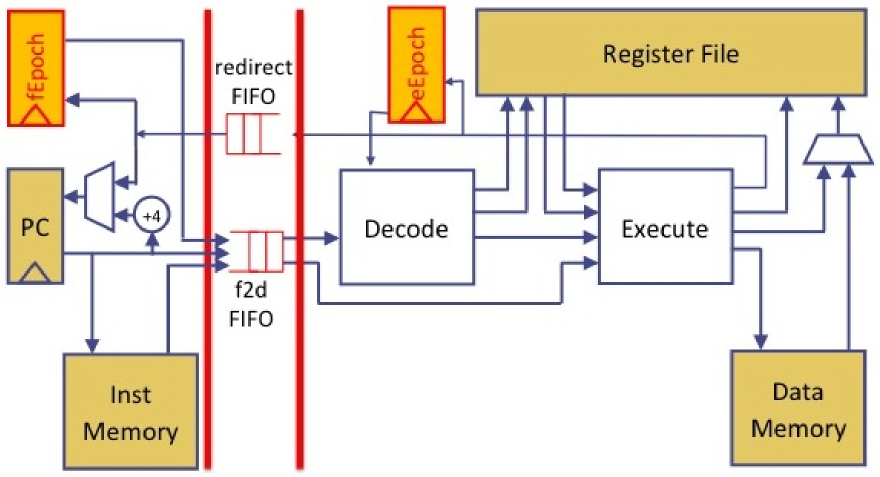


Figure 6.5: Decoupled solution

the two stages in the forward direction, and `redirect` is the reverse-direction FIFO. The Fetch stage updates `pc` and `fEpoch` based on the values it receives through `redirect`. Here is the code for this solution:

Decoupled solution

```
 1 module mkProc(Proc);
 2    FIFO #(Fetch2Execute) f2d <- mkFifo;
 3    FIFO #(Addr) execRedirect <- mkFifo;
 4    Reg #(Bool) fEpoch <- mkReg(False);
 5    Reg #(Bool) eEpoch <- mkReg(False);
 6 
 7    rule doFetch;
 8       let inst = iMem.req(pc);
 9       if (execRedirect.notEmpty) begin
10          fEpoch <= !fEpoch;  pc <= execRedirect.first;
11          execRedirect.deq;
12       end
13       else begin
14          let ppc = nextAddrPredictor (pc);
15          pc <= ppc;
16          f2d.enq (Fetch2Execute{pc:pc, ppc:ppc, inst:inst, epoch:fEpoch});
17       end
18    endrule
19 
20    rule doExecute;
21       let x = f2d.first;
22       if (x.epoch == eEpoch) begin
23          let dInst = decode(x.inst);
24          ... register fetch ...;
25          let eInst = exec(dInst, rVal1, rVal2, x.pc, x.ppc);
26          if (eInst.mispredict) begin
27             execRedirect.enq (eInst.addr);
28             eEpoch <= !eEpoch;
29          end
30       end
31       f2d.deq;
32    endrule
33 endmodule
```

Since we need only 1 bit to represent an epoch, and since epoch numbers always toggle between 0 and 1, we don't actually have to carry any epoch number on a redirect message. The very arrival of the message is a signal to toggle it. Thus, the redirect queue (`execRedirect`) only carries PC values (in later chapters, where more than two epochs may be in play simultaneously, the message will also contain an actual epoch number).

In the Fetch rule, if there is any message in the redirect queue, we update `pc` and `fEpoch` (toggle it). Otherwise, we perform the normal instruction fetch and send it into `f2d`. In the Execute rule, the conditional on line 22 effectively causes the incoming instruction to be ignored if its epoch is not equal to the actual current epoch `eEpoch`. If we resolve a misprediction we send `eInst.addr` back to the Fetch stage via the `execRedirect` queue (line 27), and immediately update the local, actual epoch `eEpoch` (line 28).

6.4 Conclusion

In this chapter we developed a decoupled 2-stage SMIPS pipeline implementation, illustrating the principle of treating the design as a distributed system of independent entities communicating only with messages over channels of unspecified latency. Decoupling simplifies reasoning about correctness, and simplifies the independent refinement of entities (stages) for higher performance. We have still been somewhat informal about what we mean by "concurrent" execution of rules (which is necessary for actually getting pipelined behavior); this will be addressed in the chapter on rule semantics.

Chapter 7

BSV Rule Semantics

Parallelism within rules, and Concurrent rules within clocks

7.1 Introduction

We now examine BSV rule semantics in more detail, which we have described only informally so far. Specifically, we will describe individual rule semantics (what it means to execute a rule by itself), and then show how these are combined to describe how all the rules of a system execute within the framework of clock cycles. With this understanding, we should be able to reason about functionality of a BSV program (what does it compute?) as well as performance (how long does it take to compute it?).

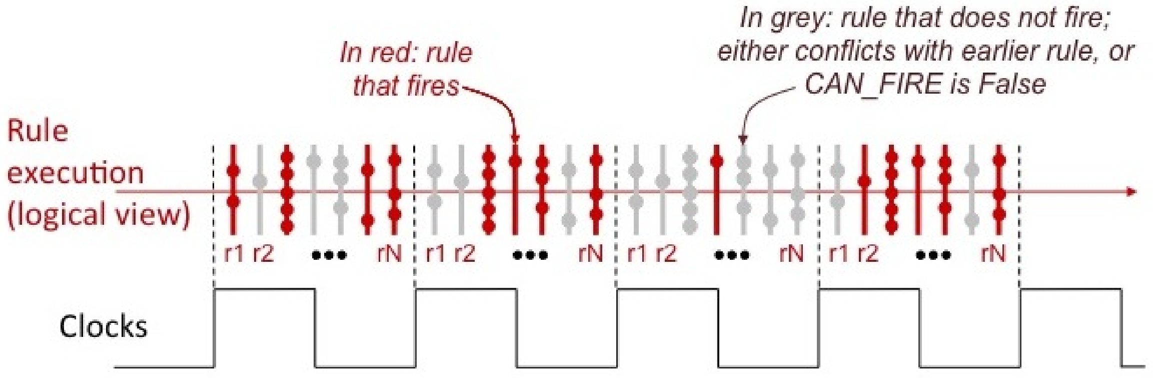

In red: rule that fires
In grey: rule that does not fire; either conflicts with earlier rule, or CAN_FIRE is False
Rule execution (logical view)
r1 r2 ••• rN
r1 r2 ••• rN
r1 r2 ••• rN
r1 r2 ••• rN
Clocks

Figure 7.1: Overview of rule semantics

Although our treatment will go bottom-up, from primitive `Action` methods to multiple actions to individual rules to multiple rules, it is useful to keep the big picture in mind, illustrated in Fig. 7.1, so that you always have a context for each topic. The figure depicts the "logical" or semantic view of the world, *i.e.,* the mental model that we keep in mind as BSV programmers. The *bsc* compiler will produce optimized hardware in which this structure may not be apparent, but its behavior will always precisely reflect this semantic model. All rules in a BSV program are placed in a *linear* order: r_1, r_2, ..., r_N (this order is chosen by the *bsc* compiler, perhaps guided by preferences from the user). The semantics are explained using this pseudo-code:

```
for each clock
   for each rule $r_j$ in $r_1$, $r_2$, ..., $r_N$ (in that order)
      let WILL_FIRE$_j$ = CAN_FIRE$_j$ AND
                          $r_j$ does not "conflict" with $r_1$, $r_2$, ..., $r_{j-1}$     (earlier rules)
      if WILL_FIRE$_j$
      then execute the body of $r_j$              (the rule "fires")
```

Fig. 7.1 shows that in each clock we consider the logical sequence of rules (each vertical bar represents a rule). On each clock, some rules fire and some don't. A rule does not fire if it either conflicts with an earlier rule in the sequence, or if its CAN_FIRE condition is false. Logically, each rule fires in an instant (zero time). Firing a rule involves performing all the actions in the rule body (depicted in the figure by circular dots on each rule).

There are many concepts in this description the require fleshing out, which we shall do in sections that follow, such as:

- What is an action in a rule?
- How do we combine multiple actions in a rule?
- What is the CAN_FIRE condition for a rule?
- When does a rule "conflict" with another?
- How does *bsc* choose a particular sequence for the set of rules?

Some terminology: we use the words "simultaneous" and "parallel" to refer to the instantaneous execution of all actions within a rule. We use the word "concurrent" to refer to the execution of multiple rules within a clock. In the linear sequence of rules, we say that r_i is "earlier" than r_j if $i < j$.

7.2 Actions and ActionValues

We have used the word "action" informally so far to refer to the actions in a rule. In fact, BSV has formal types called `Action` and `ActionValue` (the latter is a generalization of the former). The body of a rule is an expression of type `Action`, and many methods have `Action` and `ActionValue` type. Being formal types in BSV, one can even write expressions and functions of type `Action` and `ActionValue`.

The simplest action is an invocation of an `Action` method in the interface of some submodule. For methods like `enq` and `deq` for FIFOs, the method invocation is obvious in the BSV syntax, but even ordinary register writes are in fact `Action` method invocations. The syntax:

Special syntax for register reads and writes

```
rule ...
   ...
   r <= s + 1;    // register write and register read
   ...
endrule
```

is in fact just a special syntax for method invocations:

```
Register access is just method invocation
   rule ...
      ...
      r._write (s._read + 1);
      ...
   endrule
```

because BSV's standard register interface is declared as follows:

```
Register interface
interface Reg #(type t);
   method  Action   _write (t data);
   method  t        _read;
endinterface
```

i.e., a register-write is an `Action` method.

All change of state in a BSV program ultimately happens in primitive `Action` or `ActionValue` methods (the latter are just `Action`s that also return a non-void value; for simplicity, we will just talk about `Action`s from now on, keeping in mind that everything we say about `Action`s also applies to `ActionValue`s).

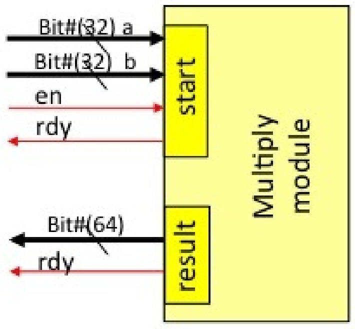


Figure 7.2: Hardware interpretation of interface methods

The hardware intuition for an `Action` method interface was shown in Fig. 3.7, which we repeat here for convenience in Fig. 7.2. In addition to input buses (method arguments), output buses (method results), and output RDY signals (method conditions), `Action` methods also have an input EN (enable) signal. Asserting this signal causes the action (state change) to take place inside the method's module. Until this signal is asserted, it does not modify any state inside the module. Performing an `Action` is synonymous with asserting the corresponding EN signal.

Each primitive module specifies the meaning of each of its methods with respect to the instant of rule firing, *i.e.,* the rule from which the method is invoked (all methods are invoked from rules, either directly or indirectly via other methods).

The idea is depicted in Fig. 7.3. Anything "read" by a method is always with respect to the state before the firing instant (due to earlier rules); we also call this the pre-`Action`

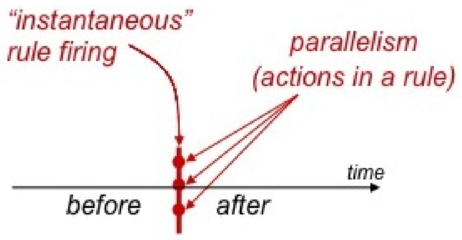


Figure 7.3: All actions in a rule happen at the same instant

state. Anything "updated" by a method produces a new state visible after the firing instant (visible to later rules); we also call this the post-`Action` state.

The primitive register module specifies that a `_read` returns the value that was in the register just before the rule execution instant (*i.e.,* from an earlier rule), and that the value written by `_write Action` is visible just after the rule execution instant (for `_reads` in later rules).

The FIFO primitive specifies that `first` returns the value at the head of the FIFO (if it is not empty), and that `enq` and `deq` change the state of the FIFO in the expected way, where this new state is visible in later rules.

7.2.1 Combining Actions

Actions are combined into larger actions with syntactic combining forms. For example:

```
x <= x + 1;
y <= y + 2;
fifo.enq (z);
```

represents a "parallel composition" of three Actions into a single composite Action. Performing the composite Action is equivalent to performing the three sub-actions. Another example:

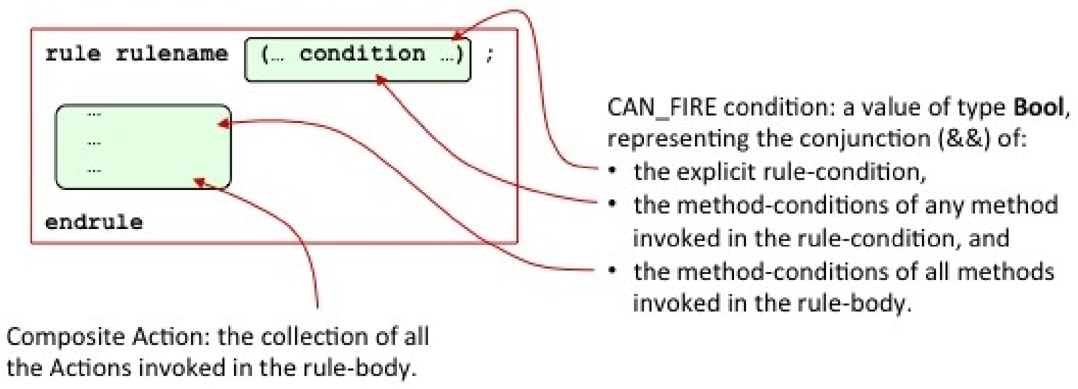
```
x <= x + 1;         // A
if (p) begin
   y <= y + 2;      // B
   fifo.enq(z);     // C
end
else
   fifoB.deq;       // D
```

This combines Actions B and C using parallel composition (call this BC); combines BC and D using a conditional composition (call this BCD), and combines A and BCD using parallel composition (call this ABCD). Performing an action created using conditional composition, like BCD, is equivalent to performing the action in either its "then" or its "else" arms, depending on the value of the conditional predicate. Note that the entire composite action ABCD is still logically instantaneous, with just a pre-`Action` state and a post-`Action` state;

there is no temporal sequencing within an `Action`, and no concept of any intermediate state within an `Action`. The entire state-change conceptually happens in a single instant.

In this way, the entire body of a rule, or of an Action method, can be viewed as an expression of type `Action`. But if we follow its recursive structure down to its component actions, and further to their sub-component actions, and so on, we eventually reach a set of `Action` method invocations.

7.3 Parallelism: semantics of a rule in isolation



rule rulename
(… condition …) ;
…
…
…
endrule
CAN_FIRE condition: a value of type **Bool**, representing the conjunction (&&) of:
• the explicit rule-condition,
• the method-conditions of any method invoked in the rule-condition, and
• the method-conditions of all methods invoked in the rule-body.
Composite Action: the collection of all the Actions invoked in the rule-body.

Figure 7.4: Anatomy of a rule

Fig. 7.4 shows the anatomy of a rule. Every rule has rule name and two semantically significant parts, highlighted in the figure as light green boxes[1]. The rule condition is a boolean expression, which may involve method invocations. The rule body is an expression of type Action which, recursively, may be composed of many sub-actions of type Action. The rule body also contains method invocations.

A rule can be executed only if its CAN_FIRE condition is true. This is a boolean condition that is the conjunction (AND) of several components:

- The value of the boolean expression in the rule condition,
- the (boolean) value of the method conditions of all methods invoked in the rule condition, and
- the (boolean) value of the method conditions of all methods invoked in the rule body.

There is a nuance to the last two components. Since a method can be invoked inside a conditional construct (`if`, `case`, ...), the method condition is qualified by the conditional predicates, so that it is only relevant if would actually be executed. For example, if we had a construct:

```
if (p) fifo1.enq (x);
else fifo2.enq (y);
```

[1]The rule name does play a role in so-called "scheduling annotations" and in debugging, but is otherwise not semantically significant; we do not discuss those features in this chapter.

then the contribution of the `fifo1.enq` and `fifo2.enq` method conditions (call them `mc1` and `mc2`) will be `(p?mc1:mc2)`, *i.e.*, `mc1` is only relevant if `p` is true, and `mc2` is only relevant if `p` is false.

If the CAN_FIRE condition of a rule is true, we say that the rule is "enabled" for firing (execution of the body). When we execute the rule (if it does not "conflict" with earlier rules), all the `Actions` in the rule (which are all typically method invocations) are performed simultaneously and instantaneously.

A rule can be seen as a specification of an acyclic combinational circuit whose inputs and outputs are connected to the interface methods of sub-modules invoked in the rule. For example, consider this rule:

Simple example for 1-rule semantics

```
rule rf (mod1.f > 3);
   x <= x+1;
   if (x != 0) y <= x >> 7;
endrule
```

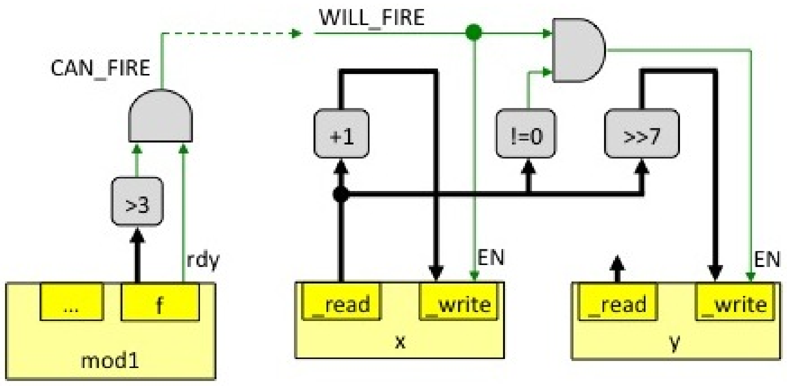


Figure 7.5: Combinational circuit for a rule

Fig. 7.5 illustrates the combinational circuit, or the "data flow graph" for this rule. The CAN_FIRE condition for this rule will be the conjunction of the expression `(mod1.f>3)` with the method conditions of the methods in the rule. The only method condition of interest is `mod1.f` since, for registers (here, `x` and `y`) the `_read` and `_write` methods have trivial method conditions (always true). The computation of the CAN_FIRE condition is seen in the AND gate on the left of the figure. Let us also call this the WILL_FIRE condition, since we are for the moment ignoring other rules in the system. The WILL_FIRE signal activates the EN input of register `x` directly. However, since the update of `y` is inside a conditional, you can see that the WILL_FIRE is further ANDed with the predicate of the conditional.

Another example:

Another example for 1-rule semantics

```
rule rg;
   let x = fifo1.first;
   if (x != 0) begin
      fifo1.deq;
      for (Integer j = 1; j <= 4; j = j + 1)
```

```
            x = x + j;
         fifo2.enq (x);
      end
   endrule
```

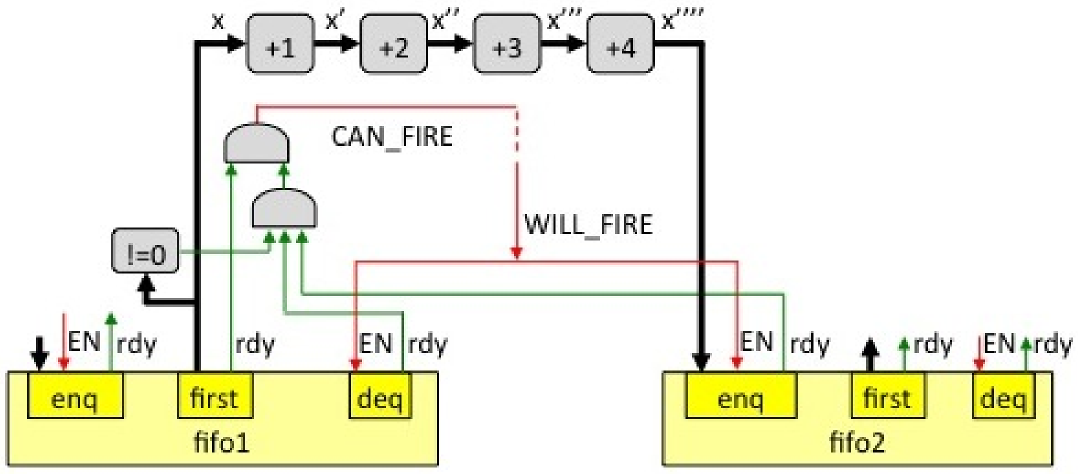


Figure 7.6: Combinational circuit for a rule: another example

Fig. 7.6 shows the data flow graph for this rule. The CAN_FIRE signal is a conjunction of the RDY signals for `fifo1.first`, `fifo1.deq` and `fifo2.enq`, but notice that the latter two are ANDed with `x!=0`, since they are only relevant if that expression is true. The CAN_FIRE becomes the WILL_FIRE signal through the schedule manager (discussed later), and this enables the `enq` and `deq` actions, which happen in one, simultaneous instant. The for-loop has been statically elaborated into a chain of four adders, incrementing by 1, 2, 3 and 4 respectively. The input to the chain comes from `fifo1.first` and the output goes into `fifo2.enq`.

Another concept reinforced in Fig. 7.6 is that ordinary variables like `x` never refer to storage locations, as they do in C or C++. They are merely names for intermediate values (in the hardware, wires). A repeated "assignment" of `x` in a statically elaborated loop, as in line 5, does not update any storage, as in C or C++: it is just conceptually naming a new `x` for another set of wires. In the figure we have disambiguated them by calling them `x'`, `x''`, `x'''` and `x''''`.

In general, the data flow graph of a rule may weave in and out of module methods, for example if an invoked method is itself a combinational function, or an invoked method is an `ActionValue` method. Further, for a user-defined sub-module, its methods will, in turn, invoke methods of deeper sub-modules. The data flow graph of a rule is the entire data flow graph, crossing user-defined module boundaries, eventually originating and terminating at methods of primitive modules.

To summarize single rule semantics:

- Compute the values of all value expressions in the rule. This is equivalent to tracing through the corresponding combinational circuit, starting with constants and method outputs and ending in method inputs.

- Compute the CAN_FIRE condition for the rule. This is a conjunction of the boolean expression explicitly given as the rule condition, along with the method conditions of all the methods mentioned in the rule. For methods inside conditionals, the method conditions are modified by the predicates of the conditionals.
- For now (single rule in isolation), let WILL_FIRE = CAN_FIRE
- Compute the EN conditions for all the action methods in the rule. For top-level methods, this is just WILL_FIRE. For methods inside conditionals, it is the conjunction of WILL_FIRE with the conditional predicates (or its negation, depending on which arm of the conditional it is in).
- Perform the actions of all enabled methods (EN is true). Thus, the overall state change when we fire a rule is composed of the state changes of the enabled Action methods when it fires.

The data flow graph is implemented by hardware combinational circuits and of course signals take time to propagate through such circuits. However, in BSV semantics we abstract away from this detail: we assume the values are instantly known on all the edges of the data flow graph, and that all the EN signals are simultaneously enabled.

7.3.1 Per-rule method well-formedness constraints

We close this section on semantics of individual rules by observing that some pairs of methods cannot be combined to execute simultaneously within the same rule.

Every primitive BSV module comes with a set of pairwise constraints on its methods, *i.e.,* for every pair of its methods m_i and m_j (including where $i = j$), whether or not that pair can both be invoked in a single rule on the same module instance. These constraints arise from hardware realizability.

For example, the primitive register modules specify that a single rule cannot invoke more than one `_write` on the same register within a rule. This makes intuitive hardware sense: if two `_write`s on a register occur at the same instant, which value should it now hold? On the other hand, there is no constraint on the number of `_read`s of a particular register that can be invoked within a rule (in hardware terms, reads are merely fan-outs of wires).

Similarly, FIFO modules have well-formedness constraints like these:

- Method `first` can be invoked multiple times.
- There can be at most one invocation of either `enq` or `deq` (but not both) on a particular FIFO in a rule.

Constraints on methods of primitive or user-defined modules may also arise due to hardware resource limitations. For example, a method `m3(x)`, when implemented in hardware, may have a single input bus for `x`; this bus can only be driven with one value at a time; thus, it would not make sense to allow two invocations of `m3` within a single rule.

Note, these constraints are on simultaneous *dynamic* invocations, which is not the same thing as the number of textual (static) occurrences of invocations. For example, it is perfectly ok to invoke `f.enq(e1)` in one arm of a conditional and `f.enq(e2)` in the other arm (on the same FIFO `f`). Even though there are two textual (static) occurrences of `f.enq()`, there can only be one dynamic invocation. The *bsc* compiler analyzes method-invocation conditions and will allow such multiple invocations if it can prove them to be mutually exclusive.

Constraints on the methods of user-defined modules follow naturally from the constraints on methods of primitive modules. For example, if a user-defined module has two methods `um1` and `um2` which both write into a common register, the constraints and meaning of the underlying `_writes` carry upward to `um1` and `um2`, that is, `um1` and `um2` cannot be invoked in the same rule.

In summary: a rule can invoke any actions for simultaneous execution, provided we respect the well-formedness constraints (mostly obvious). The *bsc* compiler will flag any violations of these constraints as static, compile-time errors.

Examples of legal and illegal combinations of actions into rules

Example 1:

Parallel methods in a rule

```
rule ra if (z>10);
   x <= x+1;
endrule
```

This rule is fine: a read and a write on register `x` is ok, as is a read from register `z`.

Example 2:

Parallel methods in a rule

```
rule rb;
   x <= x+1;
   if (p) x <= 7;
endrule
```

This rule is not ok, since it potentially performs two writes on register `x`. *bsc* will accept it only in the trivial case where it can statically prove that `p` is false.

Example 3:

Parallel methods in a rule

```
rule rb;
   if (q)
      x <= x+1;
   else
      x <= 7;
endrule
```

This rule is ok. Even though there are two syntatic occurrences of a write to `x`, they are in the context of a conditional which ensures that only one of them will be performed. It is equivalent to a more explicit muxed expression:

Parallel methods in a rule

```
rule rb;
   x <= (q ? x+1 : 7);
endrule
```

Example 4:

```
Parallel methods in a rule
rule rc;
   x <= y+1;
   y <= x+2;
endrule
```

This rule is ok, since it contains one read and one write for each of `x` and `y`. According to the given (axiomatic) semantics of the `_read` and `_write` methods, both reads of `x` and `y` read the value just before the rule execution instant, and both writes of `x` and `y` will update the registers with values visible just after the execution instant. Thus, this rule does a kind of "exchange and increment" of `x` and `y`, incrementing by 2 and 1, respectively.

Example 5:

```
Parallel methods in a rule
rule re;
   s2 = f (s1);
   s3 = g (s2);
   x <= s3;
endrule
```

This rule will be ok if it is ok to call `f` and `g` simultaneously. For example, if `f` and `g` are ordinary arithmetic functions, there is no problem. Even if `f` and `g` are the same arithmetic function, the logic will simply be replicated. However, suppose `f` and `g` contain calls to a method on some hardware module (such as reading a single port in a register file but with different indexes); then that method will have a constraint that it cannot be called simultaneously, and the rule will not be ok.

7.4 Logical semantics vs. implementation: sequential rule execution

Before we move on to the semantics of multiple rules in a clock (next section), we would like to review a concept that is ubiquitous in Computer Science, namely the clean separation of abstract, logical semantics from what happens in any particular implementation.

For example, the semantic view of a processor instruction set (such as the MIPS or x86 instruction set) is one instruction at a time. The semantics of each opcode is described in terms of what it reads and writes in registers and memory, and the semantics of a program is explained as a sequence of instructions executed one at a time. Programmers writing assembly or machine code and compilers generating such code rely on this semantic model. However, under the covers, a particular processor implementation may (and nowadays usually will) execute instructions in parallel (e.g., pipelining, and superscalar) or out-of-order, but the hardware will always behave equivalently to the sequential semantics. This separation allows clients (programmers and compilers) to create programs independently of implementors (processor designers); each can create new artifacts at their own developmental pace.

In Fig. 7.1 we described how we always view the concurrent execution of rules within a clock as a logical sequence of rule executions. This is just a programmer's view. In the actual hardware implementation generated by *bsc* there may be nothing recognizable as a sequence of rules, but the hardware will always behave equivalently to a sequential execution of rules.

7.5 Concurrent rule execution, and scheduling rules into clocks

In Sec. 7.1 and Fig. 7.1 we described BSV system execution as an operation repeated once per clock. This operation is a traversal of all the rules in the system in a logical linear sequence. For each rule, we execute it if its CAN_FIRE is true, and if it does not conflict with earlier rules in the sequence that have fired. In this section we describe the concept of inter-rule "conflict" precisely.

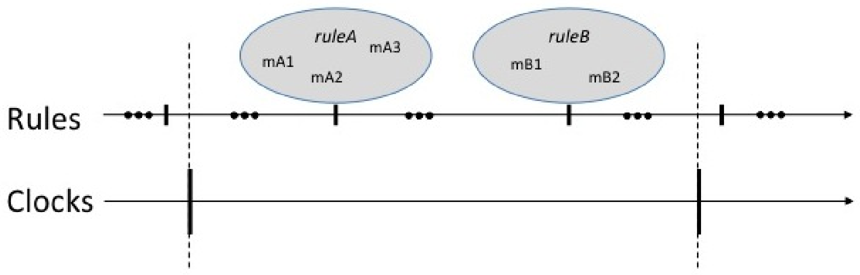


Figure 7.7: Two rules in a clock, and their methods

Conflicts arise out of violations of ordering constraints on pairs of module methods. Fig. 7.7 shows two rules *ruleA* and *ruleB* in the same clock, in that logical order. They contain method invocations `mA1`, `mA2`, `mA3`, `mB1`, and `mB2`. Of course, each rule can execute only if its guard condition is true. In addition, *ruleB* is only allowed to execute if each of its methods `mB`j is free of conflicts with any method `mA`i executed in the earlier rule.

Inter-rule method ordering constraints are specified like this:

	Allowed ordering(s)
m_i CF m_j	Either order ("conflict-free")
$m_i < m_j$ $m_j > m_i$	$\mathtt{m}_i$ must be invoked before $\mathtt{m}_j$ (else conflict) ... ditto ...
m_i C m_j	Neither order ("always conflict")

Referring back to Fig. 7.7, suppose *ruleA* executes in this clock, *i.e.,* its condition is true and it does not violate any ordering constraints with any previous rules in this clock. Then, *ruleB* executes if its condition is true and

`mA`i CF `mB`j
or `mA`i < `mB`j

Of course, *ruleB* does not execute if its condition is false. But, even if its condition is true, it cannot execute if

$$\begin{array}{ll} & \texttt{mA}i \ \mathrm{C} \ \texttt{mB}j \\ \text{or} & \texttt{mA}i > \texttt{mB}j \end{array}$$

If *ruleA* does not execute (either because its condition is false, or because its method invocations conflicted with earlier rules), the rule is irrelevant when deciding whether to execute *ruleB*.

It is important to note that the *intra*-rule constraints described earlier in Sec. 7.3.1 are completely independent of the *inter*-rule constraints described here. The former are constraints on well-formedness of a single rule, *i.e.,* if a rule can simultaneously invoke certain pairs of methods, and are checked statically by the compiler. The constraints described here are constraints on methods in different rules, and result in a dynamic decision whether to allow or suppress a rule that conflicts with an earlier one.

Where do these inter-rule constraints come frome? Once again, like intra-rule constraints, they are "given" (or axiomatic) for methods of primitive modules, and are typically imposed out of considerations of hardware realizability. Constraints on primitive methods may in turn imply constraints for methods of user-defined modules.

Consider again the register interface. For concurrent execution, we have the constraints:

`_read`	<	`_write`
`_read`	CF	`_read`
`_write`	C	`_write`

We can easily see the hardware intuitions behind these constraints. The first constraint captures the idea that, within a clock, the value we read from a register is the one from the previous clock edge, and any value we write will only be visible at the next clock edge. The constraint allows BSV register `_reads` and `_writes` to map naturally onto ordinary hardware register reads and writes. Suppose we had a rule that invoked `_write` on a particular register. Then, according to sequential rule semantics, rules later in the ordering that invoked `_read` on the same register would have to see the updated value, within the same clock, and this is not compatible with hardware semantics. Our method constraints will prevent the latter rules from firing, thus finessing this situation.

The second constraint permits multiple reads of a register within a clock, and the third constraint ensures that there is at most one write into a register in each clock.

7.5.1 Schedules, and compilation of schedules

A *schedule* is any linear sequence of rules. The pseudo-code describing rule execution semantics in Sec. 7.1 will provide correct execution for any schedule, because rules are only executed if they do not conflict with earlier rules. This raises the question: Which schedule? If there are N rules, there are $N!$ (N-factorial) possible permutations,[2] *i.e.,* schedules, so which one does *bsc* choose?

[2] In fact, more than $N!$ combinations if we allow multiple executions of a rule within a clock.

The *bsc* compiler performs extensive static analysis of the method ordering constraints on the rules of a program to choose a good schedule, one that maximizes the number of rules that can execute within each clock. The user can place "attributes" in the source code to fix certain choices in schedule. Compilation outputs allow the user to view the schedule chosen, both graphically and textually. *bsc* generates hardware that implements the rules with this schedule.

7.5.2 Examples

Let us now firm up this simple idea of rule ordering constraints by looking at a series of examples.

Example 1

Conflict-free example

```
rule ra if (z>10);
   x <= x+1;
endrule

rule rb if (z>20);
   y <= y+2;
endrule
```

The methods involved are: `z._read`, `x._read`, `x._write`, `y._read`, and `y._write`. There are no constraints between `x`'s methods and `y`'s methods and `z`'s methods. There are no constraints between `z._read` in different rules (the method is conflict free (CF) with itself). Thus, there are no constraints between the rules, and they can execute concurrently no matter which order they appear in a schedule.

Example 2

Conflict example

```
rule ra if (z>10);
   x <= y+1;
endrule

rule rb if (z>20);
   y <= x+2;
endrule
```

Here, `y._read` < `y._write` requires `ra` to precede `rb`. But, `x._read` < `x._write` requires `rb` to precede `ra`. This is a conflict, and *bsc* will ensure that these rules do not fire concurrently. In other words, whichever linear ordering *bsc* picks, it will ensure that, in any clock, the latter rule is suppressed if the former rule fires.

Example 3

Conflict-free example
```
rule ra if (z>10);
   x <= y+1;
endrule

rule rb if (z>20);
   y <= y+2;
endrule
```

Here, `y._read` < `y._write` requires `ra` to precede `rb`, and that is the only inter-rule constraint. *bsc* will try to pick a schedule where `ra` is earlier than `rb`, which will allow these rules to fire concurrently.

Example 4

Conflict example
```
rule ra;
   x <= y+1;
   u <= u+2;
endrule

rule rb;
   y <= y+2;
   v <= u+1;
endrule
```

Here, `y._read` < `y._write` requires `ra` to precede `rb`. `u._read` < `u._write` requires `rb` to precede `ra`. Once again, we have a conflict, and *bsc* will not allow these rules to fire concurrently, *i.e.,* whichever linear ordering *bsc* picks, it will ensure that, in any clock, the latter rule is suppressed if the former rule fires.

7.5.3 Nuances due to conditionals

Consider the following rules:

Conflict-freeness due to conditionals
```
rule ra;
   if (p) fifo.enq (8);
   u <= u+2;
endrule

rule rb;
   if (! p) fifo.enq (9);
   y <= y+23;
endrule
```

There are no concurrency constraints due to `u` and `y`, but what about `fifo.enq()`? The `enq` method on most FIFOs will have a concurrency constraint that it conflicts with itself, arising out of the hardware implementation constraint that we can only handle one enqueue per clock cycle. However, in the above code, the `p` and `!p` conditions ensure that, even if the rules fire concurrently, only one `enq` will happen. *bsc* takes this into account when analyzing rules for conflicts, and will in fact deem these rules to be conflict free.

However, there is a caveat: the above conclusion depends on realizing (proving) that the conditions `p` and `!p` are mutually exclusive. When those expressions get more complicated, it may not be possible for *bsc* to prove mutual exclusivity. In such cases, *bsc* is conserative and will declare a conflict due to a potential concurrent invocation of the method.

7.5.4 Hardware schedule managers

In the examples in Fig. 7.5 and Fig.7.6 we got a sense of the hardware generated for each rule in isolation. In each figure, we left unspecified (using a dashed line) the transformation of the CAN_FIRE signal into the WILL_FIRE signal. We can now fill in that last missing detail.

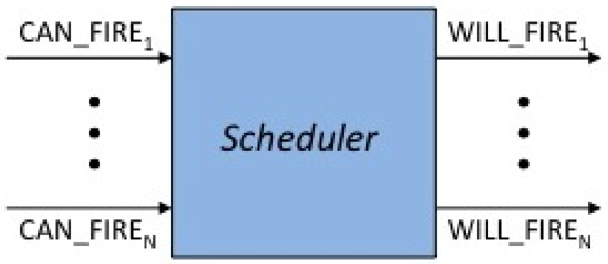


Figure 7.8: The Schedule Manager module

Conceptually, we have a module as shown in Fig. 7.8. Its inputs are the CAN_FIRE signals from all the rules, and its outputs are the WILL_FIRE signals for all the rules. It directly implements the following very simple logic:

$$\begin{aligned}
\text{WILL\_FIRE}_j = \ & \text{CAN\_FIRE}_j \ \&\& \\
& (!\ \text{WILL\_FIRE}_{i1})\ \&\& \qquad \text{(if } r_{i1} \text{ is an earlier conflicting rule)} \\
& (!\ \text{WILL\_FIRE}_{i2})\ \&\& \qquad \text{(if } r_{i2} \text{ is an earlier conflicting rule)} \\
& \text{... for each earlier conflicting rule ...}
\end{aligned}$$

In other words, this directly implements the idea that a rule is not allowed to fire if any previous conflicting rule has fired. Or, to put it another way, a rule can only fire if none of its earlier conflicting rules has fired.

7.6 Conclusion

We conclude this chapter with some very things to remember:

- Rule semantics come in two parts: intra-rule and inter-rule.
- Intra-rule semantics (parallel/simultaneous) is about execution of the rule body, which is always of type `Action`:
  - `Action` composition: sequential and conditional composition, modulo well-formedness constraints about pairs of actions that cannot be performed simultaneously.
  - Instantaneous execution of the rule body `Action` (which may be composed of other `Actions`).
  - Execution of the rule body, which transforms a pre-`Action` state into a post-`Action` state. This is logically instantaneous; there is no intermediate states.
- Inter-rule semantics (concurrency) is about the logically sequential execution of rules according to a schedule:
  - Each rule fires in a clock if its CAN_FIRE condition is true and if it does not conflict with rules that have previously fired in the same clock.
  - Its CAN_FIRE condition is a combination of the explicit rule condition expression and the method conditions of all the methods it invokes in the rule condition and body. The combination takes into account conditional constructs in the rule condition and body.
  - A rule conflicts with an earlier rule if one of its methods conflicts with a method in the earlier rule (violates a method-ordering constraint) and the earlier rule fired in this clock.
  - The *bsc* compiler picks a schedule that attempts to maximize rule concurrency.

As you write more and more BSV programs and internalize these concepts, these semantics will become as natural and second-nature to you as the traditional sequential semantics of C, C++ or your favorite sequential software language.

Chapter 8

Concurrent Components

Concurrent components, and implementing them using Concurrent Registers (CRegs)

8.1 Introduction

As we proceed to more sophisticated processor pipelines, we will repeatedly encounter the need for components that are highly concurrent, that is, modules with multiple methods that must be invoked in the same clock. The latter part of this chapter presents concurrent FIFOs (so-called "Pipeline" and "Bypass" FIFOs) which we will use heavily in our pipeline designs. Rather than presenting them as primitives we show how they can be constructed using a very useful basic component called an Concurrent Register (CReg).[1] CRegs can be used to construct other concurrent data structures as well, such as concurrent register files and register scoreboards that we will see in later chapters.

An important point is that there is a systematic methodology for designing new concurrent components—we first explicitly specify the desired concurrent semantics of the component unambiguously in terms of ordering constraints on its methods, and then implement these semantics using CRegs.

8.2 A motivating example: an up-down counter

Suppose we want to construct a module that is a two-port up-down counter of 16-bit signed integers, with the following interface:

Up-down saturating counter interface
```
interface UpDownSatCounter;
   method ActionValue #(Int #(16)) incr1 (Int #(16) delta);
```

[1]The names "Concurrent Registers" and "CRegs" are used today in BSV. The idea was originally conceived and developed by Daniel Rosenband in [10, 11] where they were called Ephemeral History Registers or EHRs.

```
   method ActionValue #(Int #(16)) incr2 (Int #(16) delta);
endinterface
```

Let's assume that a module with this interface has some internal state holding the `Int#(16)` counter value, initially 0. When either `incr` method is called, the internal value should change by `delta` (which may be negative), and the the previous value is returned. Here is a module implementing this:

Up-down saturating counter module

```
module mkUpDownCounter (UpDownSatCounter);

   Int #(16) ctr <- mkReg (0);

   function ActionValue #(Int #(16)) fn_incr (Int #(16) delta);
      actionvalue
         ctr <= ctr + delta;
         return ctr;    // note: returns previous value
      endactionvalue
   endfunction

   method ActionValue #(Int #(16)) incr1 (Int#(16) delta) = fn_incr (delta);

   method ActionValue #(Int #(16)) incr2 (Int#(16) delta) = fn_incr (delta);
endmodule
```

We define an internal function `fn_incr` that performs the desired increment and returns the previous value, and then we define the two methods to just invoke this function. In the function, note that even though the register updates are textually before the `return ctr` statement, we return the previous value because an Action or ActionValue in BSV is semantically an instantaneous event, and all "reads" return the previous value.

8.2.1 Intra-clock concurrency and semantics

The module above may be functionally correct, but in hardware design we are usually interested in more than that—performance, in particular. The performance of a system that uses this up-down counter may boil down to this question: can `incr1` and `incr2` be operated in the same clock cycle, *i.e.,* concurrently? For example, the counter may be used for credits in a network flow-control application, incremented whenever a packet is sent and decremented whenever a packet is received. If both methods cannot be called concurrently, we may not be able to send and receive a packet on the same clock.

If we compile the code as written with *bsc* we will discover that the two methods cannot be operated in the same clock cycle. This follows from our discussion of conflicts in Sec. 7.5. Both methods invoke methods `ctr._read` and `ctr._write`; the method constraints require that `incr1` precedes `incr2` and vice versa, a classic case of a conflict, thereby preventing concurrent execution.

Before we try solving this, we must first examine a crucial semantic issue: what does it mean for these methods to be operated concurrently in the same cycle? Specifically, what do we mean by the "previous value" returned by each method? Suppose the current counter value is 3 and we invoke `incr1(5)` and `incr2(-7)` concurrently (in the same clock). In pure rule semantics, where each rule is atomic, either `incr1(5)` happens before `incr2(-7)` or vice versa. In the former ordering, `incr1(5)` should return 3 and `incr2(-7)` should return 8 ($= 3+5$). In the latter ordering, `incr2(-7)` should return 3 and `incr1(5)` should return -4 ($= 3-7$). These choices are not inherently right or wrong—the designer should choose one based on the application's requirements:

	`incr1` < `incr2`	(A)
or	`incr2` < `incr1`	(B)

Below, we will show that we can implement either choice with CRegs.

8.2.2 Concurrent Registers (CRegs)

As we saw in the previous section, using ordinary registers in the implementation does not permit the desired concurrency. For this, we use a somewhat richer primitive called a CReg (for Concurrent Register) which will have n logically sequenced `Reg` interfaces for reads and writes. In fact, the conventional ordinary register is just a special case of a CReg with just one `Reg` interface.

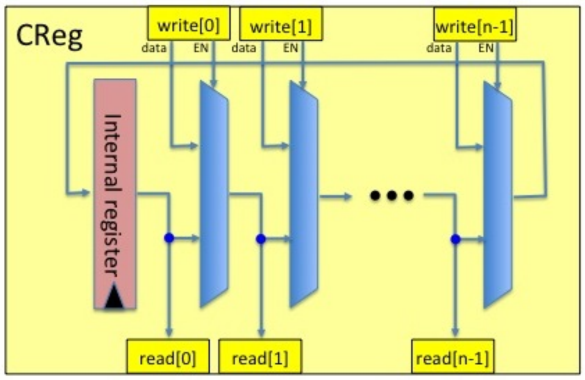


Figure 8.1: A Concurrent Register (CReg) implementation

A CReg is a module whose interface is an array of n `Reg` interfaces, indexed by $0 \leq j \leq n-1$, and which can all be read and written concurrently (in the same clock). We'll call each `Reg` sub-interface a "port". Fig. 8.1 illustrates a possible CReg implementation (other implementations are possible, too). Suppose we read from port j, and suppose zero or more writes were performed on ports $< j$. Then the value returned on read port j is the "most recent" write, *i.e.,* corresponding to the highest port $< j$ amongst the writes. If there are no writes into any ports $< j$, the value returned is the value originally in the register. Finally, the register is updated with the value written at the highest port number, if any; if there are no writes, the original value is preserved. Thus, increasing port numbers correspond to an intra-clock, concurrent, logical sequence.

The intra-rule (simultaneous/parallel) semantics of each `Reg` sub-interface `cr[`j`]` of a CReg `cr` is just like an ordinary register:

- `cr[`j`]._read` and `cr[`j`]._write` can be invoked in the same Action;
- if so, then the `_read` returns the pre-Action value and the `_write`'s value is visible post-Action.

The inter-rule (concurrent) semantics on a CReg `cr`'s ports are:

$$\begin{array}{ll}
\quad\texttt{cr[0].\_read} & < \texttt{cr[0].\_write} \\
< \texttt{cr[1].\_read} & < \texttt{cr[1].\_write} \\
< \ldots & < \ldots \\
< \texttt{cr[}n-1\texttt{].\_read} & < \texttt{cr[}n-1\texttt{].\_write}
\end{array}$$

Specifically, note that `cr[`i`]._write` precedes `cr[`j`]._read` if $i < j$; this is precisely what makes a value written by a `._write` visible to a `._read` within the same clock (which was not possible with an ordinary register).

8.2.3 Implementing the counter with CRegs

Using CRegs, it is easy to implement our up-down counter.

Up-down saturating counter with CRegs

```
1  module mkUpDownCounter (UpDownSatCounter);
2
3     Reg #(Int#(16)) ctr [2] <- mkCReg (2, 0); // 2 ports, initial value 0
4
5     function ActionValue#(Int#(16)) fn_incr (Integer port, Int#(16) delta);
6        actionvalue
7           ctr [port] <= ctr [port] + delta;
8           return ctr [port];
9        endactionvalue
10    endfunction
11
12    method ActionValue#(Int#(16)) incr1 (Int#(16) delta) = fn_incr(0,delta);
13
14    method ActionValue#(Int#(16)) incr2 (Int#(16) delta) = fn_incr(1,delta);
15 endmodule
```

In line 3, we instantiate a CReg holding an `Int#(16)` value, with two ports (the "`[2]`" declarator on `ctr` and the "2" parameter to `mkCReg ()`). The function `fn_incr` is parametrized by by a port number. The method `incr1` uses port 0, while the method `incr2` uses port 1. This implements semantic choice (A), *i.e.,* `incr1<incr2`. If we exchanged the 0 and 1 indexes, we'd implement choice (B), *i.e.,* `incr2<incr1`. These methods can indeed be invoked concurrently, *i.e.,* in the same clock.

Exercise

Add a third method which just sets the counter to a known value n:

Additional set method
```
   method Action set (UInt #(4) n);
```

First, define a semantics for this, *i.e.,* provide a specification of what should happen when this method is called in the same clock as either or both of the other two methods (there is no "right" answer to this: you should choose a semantics). Then, extend the module implementation so that it implements this new method with these semantics.

8.3 Concurrent FIFOs

In Sec. 4.3 we introduced elastic pipelines where the stages are separated by FIFOs. For true pipeline behavior, all the individual rules (one for each stage) must be capable of firing concurrently, so that the whole pipeline moves together. More specifically, the `enq` method in an upstream rule must fire concurrently with the `first` and `deq` methods in a downstream rule. In this section we will introduce various FIFOs with this behavior, but with different semantics. Most of the FIFOs we present will have the following standard interface from the BSV library, which is obtained by importing the `FIFOF` package:

FIFOF interface
```
interface FIFOF #(type t);
   method Bool notEmpty;
   method Bool notFull;

   method Action enq (t x);

   method t      first;
   method Action deq;

   method Action clear;
endinterface
```

The `notEmpty` and `notFull` methods are always enabled, and can be used to test for emptiness and fullness. The `enq` method will only be enabled in the `notFull` state. The `first` and `deq` methods will only be enabled in the `notEmpty` state. The `clear` method is always enabled, and can be used to set the FIFO to the empty state.

8.3.1 Multi-element concurrent FIFOs

It is subtle, but not hard to create a concurrent multi-element FIFO using ordinary registers. Here is a first attempt at creating a two element FIFO (note that arithmetic on `Bit#(n)` types is unsigned, and will wrap around):

Two element FIFO, first attempt

```
module mkFIFOF2 (FIFOF #(t));
   Reg #(t)          rg_0     <- mkRegU;      // data storage
   Reg #(t)          rg_1     <- mkRegU;      // data storage
   Reg #(Bit #(1))   rg_tl    <- mkReg (0);   // index of tail (0 or 1)
   Reg #(Bit #(2))   rg_count <- mkReg (0);   // # of items in FIFO

   Bit #(2) hd = extend (rg_tl) - rg_count;   // index of head

   method Bool notEmpty = (rg_count > 0);
   method Bool notFull  = (rg_count < 2);

   method Action enq (t x) if (rg_count < 2);
      if (rg_tl == 0)
         rg_0 <= x;
      else
         rg_1 <= x;
      rg_count <= rg_count + 1;
   endmethod

   method t first () if (rg_count > 0);
      return ((hd[0] == 0) ? rg_0 : rg_1);
   endmethod

   method Action deq () if (rg_count > 0);
      rg_count <= rg_count - 1;
   endmethod

   method Action clear;
      rg_count <= 0;
   endmethod
endmodule
```

Although functionally ok, `enq` and `deq` cannot run concurrently because they both read and write the `rg_count` register, and so any concurrent execution would violate a "`_read` < `_write`" constraint. We can modify the code to avoid this, as follows:

Concurrent two element FIFO

```
module mkFIFOF2 (FIFOF #(t));
   Reg #(t)          rg_0  <- mkRegU;      // data storage
   Reg #(t)          rg_1  <- mkRegU;      // data storage
   Reg #(Bit #(2))   rg_hd <- mkReg (0);   // index of head
   Reg #(Bit #(2))   rg_tl <- mkReg (0);   // index of tail

   Bit #(2) count = ( (rg_hd <= rh_tl)
                    ? (rg_tl - rg_hd)
                    : (rg_tl + 2 - rg_hd));

   method Bool notEmpty = (count > 0);
   method Bool notFull  = (count < 2);

   method Action enq (t x) if (count < 2);
      if (rg_tl[0] == 0)
         rg_0 <= x;
      else
         rg_1 <= x;
      rg_tl <= rg_tl + 1;
   endmethod

   method t first ()  if (count > 0);
      return ((rg_hd[0] == 0) ? rg_0 : rg_1);
   endmethod

   method Action deq () if (count > 0);
      rg_hd <= rg_hd + 1;
   endmethod

   method Action clear;
      rg_hd <= 0;
      rg_tl <= 0;
   endmethod
endmodule
```

In this version of the code, `enq` and `deq` both read the `rg_hd` and `rg_tl` registers. The former only writes `rg_tl`, and the latter only writes `rg_hd`. Thus, the methods can run concurrently. Note that `first` reads `rg_0` and `rg_1`, while `enq` writes them; therefore, because of the "`_read` < `_write`" constraint, `first` must precede `enq` in any concurrent schedule.

Larger FIFOs

In the two-element FIFOs just shown, we explicitly instantiated two registers to hold the FIFO data. This would get quite tedious for larger FIFOs, where $n > 2$. We could, instead, use BSV's `Vector` library, like this:

Concurrent n-element FIFO

```
import Vector :: *;
...
module mkFIFOF_n #(numeric type n)
                 (FIFOF #(t));
   Vector #(n, Reg #(t)) vrg <- replicateM (mkRegU);
   ...
   method t first ()  if (count > 0);
      return vrg [rg_hd];
   endmethod
   ...
endmodule
```

The BSV library provides the `mkSizedFIFOF` family of modules to create such FIFOs. When n is very large, even such implementations would not be efficient, since it requires large muxes to select a particular value from the vector. The BSV library provides various memory-based FIFO modules for very large FIFOs, in the `BRAMFIFO` package.

8.3.2 Semantics of single element concurrent FIFOs

If we attempt to define a single element concurrent FIFO using ordinary registers, we will face a problem. Here is a first attempt (a specialization of the two element FIFO):

One element FIFO, first attempt

```
module mkFIFOF1 (FIFOF #(t));
   Reg #(t)         rg       <- mkRegU;     // data storage
   Reg #(Bit #(1))  rg_count <- mkReg (0);  // # of items in FIFO (0 or 1)

   method Bool notEmpty = (rg_count == 1);
   method Bool notFull  = (rg_count == 0);

   method Action enq (t x) if (rg_count == 0);
      rg <= x;
      rg_count <= 1;
   endmethod

   method t first ()  if (rg_count == 1);
      return rg;
   endmethod

   method Action deq () if (rg_count == 1);
      rg_count <= 0;
   endmethod

   method Action clear;
      rg_count <= 0;
   endmethod
endmodule
```

The problem is that, since a one element FIFO is either empty or full (equivalently, either `notFull` or `notEmpty`), either `enq` or `deq` will be enabled, but not both. Thus, we could never concurrently `enq` and `deq` into this FIFO.

Before looking at implementations, let us first step back and clarify what we mean by concurrency on a one element FIFO, *i.e.,* define the concurrency semantics. There are two obvious choices:

- In a *Pipeline FIFO*, we can `enq` and `deq` concurrenty even when it already contains an element, with the following logical ordering:
  `first` < `enq`
  `deq` < `enq`
  This ordering makes it clear that the element returned by `first` is the element already in the FIFO; that `deq` logically empties the FIFO; and that `enq` then inserts a new element into the FIFO. When the FIFO is empty, `first` and `deq` are not enabled, and there is no concurrency.

- In a *Bypass FIFO*, we can `enq` and `deq` concurrenty even when it contains no elements, with the following logical ordering:
  `enq` < `first`
  `enq` < `deq`
  This ordering makes it clear that `enq` inserts a new element into the empty FIFO, and this is the element returned by `first` and emptied by `deq`. Thus, the enqueued element can "fly through" the FIFO within the same clock, which is why it is called a Bypass FIFO. When the FIFO is full, `enq` is not enabled, and there is no concurrency.

As we shall see in subsequent chapters, both kinds of FIFOs are used heavily in processor design. Typically, Pipeline FIFOs are used in the forward direction of a pipeline, and Bypass FIFOs are used for feedback paths that go upstream against the pipeline data flow. Fig. 8.2 illustrates a typical structure. For concurrent operation we typically want rule rb to be

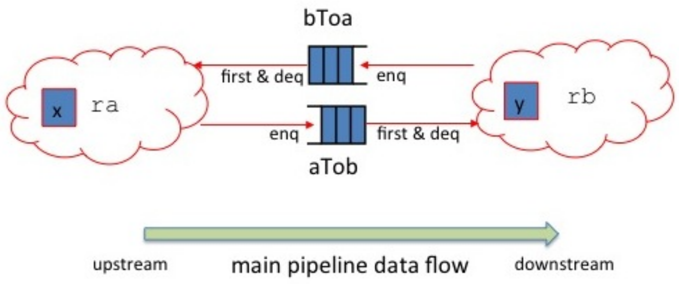


Figure 8.2: Typical Pipeline and Bypass FIFO usage

earlier in the schedule than rule ra because when the forward FIFO aTob has data we want, conceptually to dequeue it first from rb and then concurrently refill it from ra. Thus, for a consistent schedule, you can see that:

`aTob.first` and `aTob.deq`	<	`aTob.enq`
`bToa.enq`	<	`bToa.first` and `bToa.deq`

and thus aTob is a Pipeline FIFO and bToa is a Bypass FIFO. There are other possibilities, namely that one or both of the FIFOs is a "conflict-free" FIFO with no ordering constraint on `first/deq` and `enq`.

A nuance: although single element FIFOs make the concurrency question stark, the same issues can arise even in larger element FIFOs ($n > 1$). In those FIFOs this issue is often masked because the FIFO has additional states where it is neither empty nor full, and in those states, concurrency is easy. But even in those FIFOs, we may want to enrich the corner cases to have Pipeline FIFO behavior when it is actually full, or Bypass FIFO behavior when it is actually empty.

8.3.3 Implementing single element concurrent FIFOs using CRegs

With CRegs, it is easy to implement single element Pipeline and Bypass FIFOs. Here is an implementation of a Pipeline FIFO:

Pipeline FIFO with CRegs

```
module mkPipelineFIFOF (FIFOF #(t));

   Reg #(t)          cr[3]       <- mkCRegU (3);    // data storage
   Reg #(Bit #(1)))  cr_count[3] <- mkCReg (3, 0);  // # of items in FIFO

   method Bool notEmpty = (cr_count[0] == 1);
   method Bool notFull  = (cr_count[1] == 0);

   method Action enq (t x) if (cr_count[1] == 0);
      cr[1] <= x;
      cr_count[1] <= 1;
   endmethod

   method t first ()  if (cr_count[0] == 1);
      return cr[0];
   endmethod

   method Action deq () if (cr_count[0] == 1);
      cr_count[0] <= 0;
   endmethod

   method Action clear;
      cr_count[2] <= 0;
   endmethod
endmodule
```

We've generalized the data storage and item count from ordinary registers to CRegs. The `notEmpty`, `first` and `deq` methods use port 0 of the CRegs, and are thus earliest in the schedule. The `notFull` and `enq` methods use port 1 of the CRegs, and are thus next in the schedule. Finally, the `clear` method uses port 2, and is thus last in the schedule.

Note the choice of indexes in the definitions of `notEmpty` and `notFull`. This is because `notEmpty` is often used in a rule condition whose body invokes `first` and `deq`; hence they all use the same CReg index (0). If we used different indexes, the state of the FIFO could change (due to some intervening rule) between the `notEmpty` test and the `first`/`deq` invocations, which would be surprising to the user. Similarly, `notFull` is often used in a rule condition whose body invokes `enq`; hence they both use the same CReg index (1).

Similarly, here is an implementation of a Bypass FIFO:

Bypass FIFO with CRegs

```
module mkBypassFIFOF (FIFOF #(t));
   Reg #(3, t)        cr[3]       <- mkCRegU (3);   // data storage
   Reg #(Bit #(1)))  cr_count[3] <- mkCReg (3,0);  // # of items in FIFO

   method Bool notEmpty = (cr_count[1] == 1);
   method Bool notFull  = (cr_count[0] == 0);

   method Action enq (t x) if (cr_count[0] == 0);
      cr[0] <= x;
      cr_count[0] <= 1;
   endmethod

   method t first ()  if (cr_count[1] == 1);
      return cr[1];
   endmethod

   method Action deq () if (cr_count[1] == 1);
      cr_count[1] <= 0;
   endmethod

   method Action clear;
      cr_count[2] <= 0;
   endmethod
endmodule
```

Here we have just reversed the port assignments compared to the Pipeline FIFO: `notFull` and `enq` use port 0 and are thus earliest in the schedule; `notEmpty`, `first` and `deq` use port 1 and are next in the schedule; and `clear`, as before, used port 2 and is last in the schedule.

Chapter 9

Data Hazards (Read-after-Write Hazards)

9.1 Read-after-write (RAW) Hazards and Scoreboards

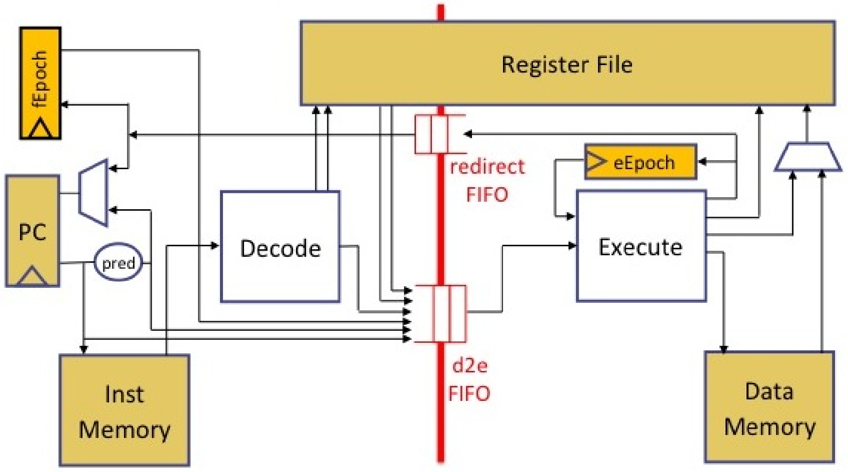


Figure 9.1: Alternate two-stage pipelining of SMIPS

Consider an alternative partitioning of our 2-stage pipeline from Chapter 6. Let us move the stage boundary from its current position, between the fetch and decode functions, to a new position, between the decode and execute functions, as illustrated in Fig. 9.1. The decode state looks up the values of the source registers in the Register File, and these values are forwarded to the Execute stage. The information carried from the Decode to the Execute stage (in the `d2e` FIFO) has the following data type:

Decode to Execute information
```
typedef struct {
   Addr  pc;
   Addr  ppc;                // predicted PC
   Bool  epoch;
   DecodedInst dInst;
   Data rVal1;                 // value from source register 1
   Data rVal2;                 // value from source register 2
} Decode2Execute
  deriving (Bits, Eq);
```

The code for the processor now looks like this:

Processor pipelined at Decode-to-Execute

```
module mkProc(Proc);
   Reg#(Addr)        pc <- mkRegU;
   RFile             rf <- mkRFile;
   IMemory           iMem <- mkIMemory;
   DMemory           dMem <- mkDMemory;

   FIFO #(Decode2Execute) d2e <- mkFIFO;    // forward

   Reg#(Bool)    fEpoch <- mkReg(False);
   Reg#(Bool)    eEpoch <- mkReg(False);
   FIFO#(Addr)   execRedirect <- mkFIFO;    // backward

   rule doFetch;
      let inst = iMem.req(pc);
      if (execRedirect.notEmpty) begin
         fEpoch <= ! fEpoch;
         pc <= execRedirect.first;
         execRedirect.deq;
      end
      else begin
         let ppc = nextAddrPredictor(pc);
         pc <= ppc;
         // ---- The next three lines used to be in the Execute rule
         let dInst = decode(inst);
         let rVal1 = rf.rd1 (validRegValue(dInst.src1));
         let rVal2 = rf.rd2 (validRegValue(dInst.src2));
         d2e.enq (Decode2Execute {pc: pc, ppc: ppc,
                                  dIinst: dInst, epoch: fEpoch,
                                  rVal1: rVal1, rVal2: rVal2});
      end
   endrule

   rule doExecute;
      let x = d2e.first;
      let dInst = x.dInst;
      let pc    = x.pc;
      let ppc   = x.ppc;
      let epoch = x.epoch;
      let rVal1 = x.rVal1;
      let rVal2 = x.rVal2;
      if (epoch == eEpoch) begin
         ... same as before ...

         ... rf.wr (validRegValue (eInst.dst), eInst.data);
         if (eInst.mispredict) begin
            ... same as before ...
         end
      end
      d2e.deq
   endrule
endmodule
```

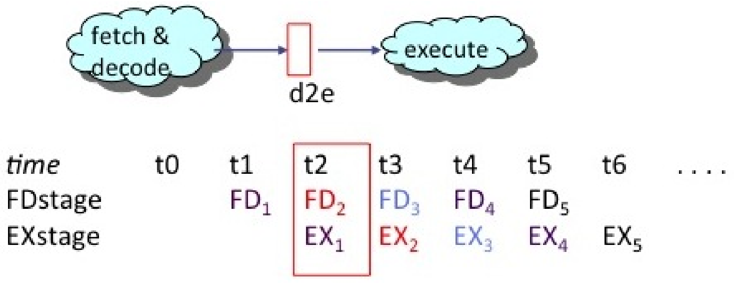


Figure 9.2: Classical pipeline schedule

This code is not quite correct, since it has a "data hazard." The Fetch rule reads from the register file (`rf.rd1` and `rf.rd2`), and the Execute rule writes to the register file (`rf.wr`). Unfortunately the Fetch rule may be reading the wrong values from the register file ("stale values"). This can be understood using Fig. 9.2, which is a classical illustration of a pipeline schedule (*i.e.,* a sketch of what happens in each pipe stage at each point in time). At time t1, the Fetch-Decode stage is occupied by the first instruction, depicted by FD_1. At time t2, this instruction now occupies the Execute stage, depicted by EX_1, while the next instruction now occupies the Fetch-Decode stage, depicted by FD_2, and so on. Suppose these two instructions were:

I_1	R1 = Add (R2, R3)
I_2	R4 = Add (R1, R2)

When I_2 reads register R1 (in FD_2), it should get the value computed by I_1 (in EX_1). However, in the schedule above, FD_2 and EX_1 occur at the same time t2. The EX_1 register update will not be visible until time t3, and so FD_2 will read an old (stale) value. This is called a Read-after-Write (or RAW) data hazard. What we need to do is to delay, or "stall" I_2 until I_1 has written its value into the register file. This is illustrated in Fig. 9.3. FD_2 is stalled until t3, when it can read the correct values. This also means that the execute stage does nothing during t3; we refer to this as a "bubble" in the pipeline.

To deal with RAW hazards, we need to:

- Keep a pending list of the names of those registers which will be written by instructions that are still ahead in the pipeline (in the Execute stage). When an instruction enters

time	t0	t1	t2	t3	t4	t5	t6	. . . .
FDstage		FD$_1$	FD$_2$	FD$_2$	FD$_3$	FD$_4$	FD$_5$	
EXstage			EX$_1$	—	EX$_2$	EX$_3$	EX$_4$	EX$_5$

Figure 9.3: Pipeline schedule with a stall

the Execute stage we enter its destination in this list; when the instruction completes, we remove this entry.

- Stall the Decode stage when it has an instruction with a RAW hazard, i.e., whose source registers are in the pending list.

In computer architecture literature, this pending list is called a “Scoreboard.” In BSV, we express its interface as follows:

```
Scoreboard interface
interface Scoreboard;
   method Action insert (Maybe #(RIndx) dst);
   method Bool search1 (Maybe #(RIndx) src1);
   method Bool search2 (Maybe #(RIndx) src2);
   method Action remove;
endinterface
```

The `search1`, `search2` and `insert` methods are used by the Decode stage, and the `remove` method is used by the Execute stage. The `search1` method searches the scoreboard to see if the `src1` register name exists in the scoreboard. The `search2` method has the same functionality; we need two such methods because we need to test both source registers of an instruction in parallel. The `insert` method adds a new destination register name into the scoreboard. The `remove` method discards the oldest entry in the scoreboard (the one that was `insert`ed first). Conceptually, the scoreboard is a FIFO, with `insert` corresponding to the enqueue operation and `remove` the dequeue operation.

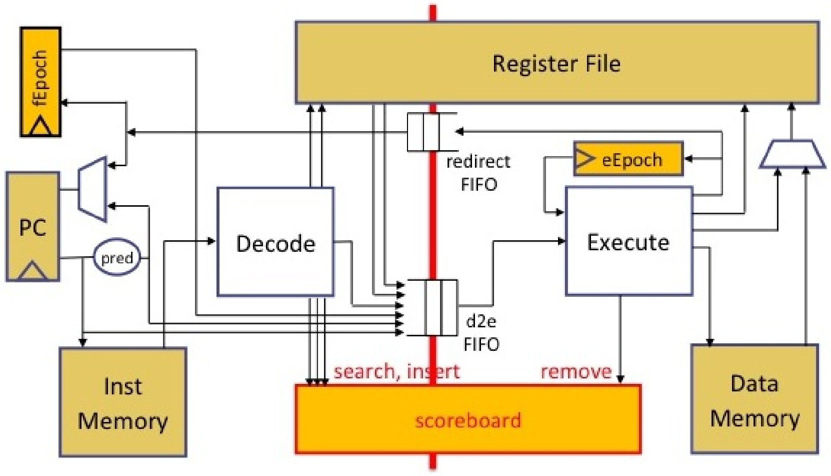


Figure 9.4: Pipeline with scoreboard

Fig. 9.4 illustrates the pipeline along with the scoreboard. We modify the processor code as follows.

Scoreboard interface

```
1
2  module mkProc(Proc);
3    .. same state elements as before ...
4
5     Scoreboard sb <- mkScoreboard (1);
6
7     rule doFetch;
8        let inst = iMem.req(pc);
9        if (execRedirect.notEmpty) begin
10          ... seame as before ...
11       end
12       else begin
13          let ppc = nextAddrPredictor (pc);
14          let dInst = decode(inst);
15          let stall = sb.search1 (dInst.src1) || sb.search2 (dInst.src2);
16          if (! stall) begin
17             let rVal1 = rf.rd1 (validRegValue(dInst.src1));
18             let rVal2 = rf.rd2 (validRegValue(dInst.src2));
19             d2e.enq(Decode2Execute {pc: pc, ppc: ppc,
20                                     dIinst: dInst, epoch: fEpoch,
21                                     rVal1: rVal1, rVal2: rVal2});
22             sb.insert(dInst.rDst);
23             pc <= ppc;
24          end
25       end
26    endrule
27
28    rule doExecute;
29       let x = d2e.first;
30       ... same as before ...
31       if (epoch == eEpoch) begin
32          ... same execution as before ...
33       end
34       d2e.deq;
35       sb.remove;
36    endrule
37 endmodule
```

In line 5 we instantiate a scoreboard with size 1 (holds one entry). This is because, in this pipeline, there is at most one pending instruction in the Execute stage. In line 15 we search the scoreboard for data hazards; we perform the register reads and forward the instruction in the following lines only if we do not stall. In line 22 we insert the destination register of the current instruction into the scoreboard. Note that line 23 updates the PC only if we do not stall; if we stall, this rule will be repeated at the same PC. In the Execute rule, the only difference is that, in line 35, we remove the completed instruction's destination register from the scoreboard.

We invoke `search1`, `search2` and `insert` in a single rule (`doFetch`), i.e., at a single instant. Clearly, we expect the single-rule semantics of these methods to be such that the searches check the *existing* state of the scoreboard (due to previous instructions only), and `insert` creates the next state of the scoreboard. There is no concurrency issue here since they are invoked in the same rule.

9.2 Concurrency issues in the pipeline with register file and scoreboard

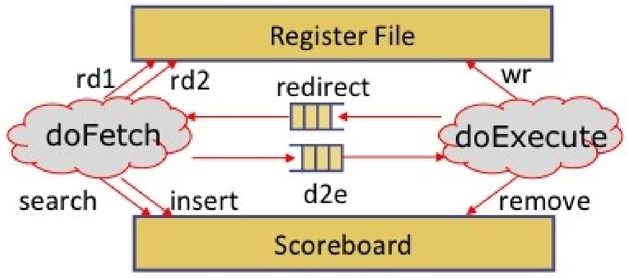


Figure 9.5: Abstraction of the pipeline for concurrency analysis

Fig.9.5 is an abstraction of the key elements of the pipeline so that we can analyze its concurrency.

In general, we think of the concurrent schedule of pipelines as going from right to left (from downstream to upstream). Please refer back to Fig. 4.6 which depicts an elastic pipeline with a number of stages. In full-throughput steady state, it is possible that every pipeline register (FIFO) is full, i.e., contains data. In this situation, the only rule that can fire is the right-most (most downstream) rule, because it is the only one that is not stalled by a full downstream FIFO. Once it fires, the pipeline register to its left is empty, and this permits the rule to its left to fire. Its firing, in turn, permits the rule to its left to fire, and so on. Of course, these are concurrent firings, i.e., they all fire in the same clock with the right-to-left logical order, thus advancing data through the entire pipeline on each clock.

This idea was also discussed around Fig. 8.2, where we saw that we typically deploy PipelineFIFOs in the forward direction (with `first` and `deq` < `enq`) and BypassFIFOs in the reverse direction (with `enq` < `first` and `deq`).

Referring now again to Fig.9.5, we expect a schedule with rule `doExecute` followed by rule `doFetch`, `d2e` is PipelineFIFO, and `redirect` is a BypassFIFO. Consistent with this schedule:

- We expect Scoreboard to have `remove` < `insert`. This is just like a PipelineFIFO, whose implementation we have already seen in Sec. 8.3.3.
- We expect Register File to have `wr` < `rd1` and `rd2`. This read-after-write ordering is like a BypassFIFO, and is the opposite order of a conventional register (where `_read` < `_write`).

A register file with the given method ordering can easily be implemented using CRegs (described in Sec. 8.2.2):

Register File with R-A-W
```
module mkBypassRFile(RFile);
   Vector #(32, Array #(Reg #(Data))) rfile <- replicateM (mkCReg (2, 0));

   method Action wr (RIndx rindx, Data data);
      if (rindex !== 0)
         (rfile [rindex]) [0] <= data;
   endmethod

   method Data rd1 (RIndx rindx) = (rfile [rindx]) [1];

   method Data rd2 (RIndx rindx) = (rfile [rindx]) [1];
endmodule
```

9.3 Write-after-Write Hazards

What happens if a fetched instruction has no RAW hazards, but writes to the same register as an instruction that is still in the pipeline? This is called a Write-after-Write (WAW) hazard. Note, there can be no instruction ahead of it that reads that register, since any instruction would still be stalled in the Fetch stage.

There are two ways we could solve this problem:

1. If we implement our scoreboard as a FIFO, there is no problem. The current instruction will be enqueued into the scoreboard, which may now contain two identical entries. Any later instruction that reads this register will be stalled until both corresponding writing instructions have completed and these entries have been removed.
2. We could implement the scoreboard as a simple bit vector with one bit per register in the register file. The `insert` method sets a bit, and the `remove` method resets it. In this case, since we can't keep track of multiple instructions writing to the same register, we must stall any instruction that has a WAW hazard. Thus, the Decode stage, in addition to testing if the source registers are in the scoreboard, must now also test the destination register. This will require another `search3` method in the scoreboard module.

Note: the `search1`, `search2` and `search3` methods can be combined into a single method taking multiple arguments.

9.4 Deeper pipelines

Fig. 9.6 illustrates a version of our pipeline that is partitioned into 3 stages. The previous Execute stage has been refined into two parts, one that executes the ALU and memory

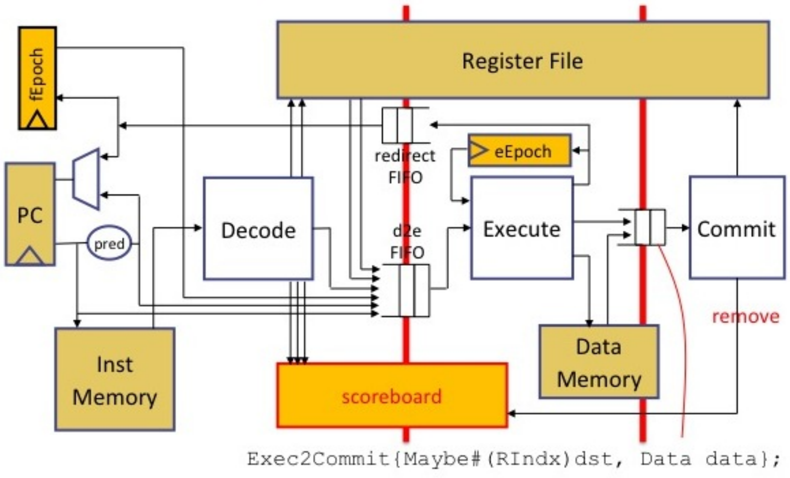


Figure 9.6: 3-stage pipeline

operations and another, the "Commit" stage, that performs the writeback of the output value to the register file and the remove operation on the scoreboard (if the instruction has an output). Since there can now be two instructions "in flight" ahead of the Decode stage, we need to increase the depth of the scoreboard queue to hold at least two destination register names.

In the Execute stage, when we kill a wrong-path instruction, we can pass an "Invalid" destination to the Commit stage, like so:

Execute stage in 3-stage pipeline
```
rule doExecute;
   ... same as before ...
   if (epoch == eEpoch) begin
      ... same as before, but without rf.wr() ...
      e2c.enq (Exec2Commit {dst:eInst.dst, data:eInst.data});

   else
      e2c.enq (Exec2Commit {dst:Invalid, data:?});    // will be ignored
    d2e.deq;
endrule

rule doCommit;
   let dst = eInst.first.dst;
   let data = eInst.first.data;
   if (dst matches tagged Valid .rdst) begin
      rf.wr (rdst, data);
      sb.remove;    // was in doExecute
   end
   e2c.deq;
endrule
```

The new `doCommit` rule performs the writeback, if the destination is valid, and also removes the destination from the scoreboard.

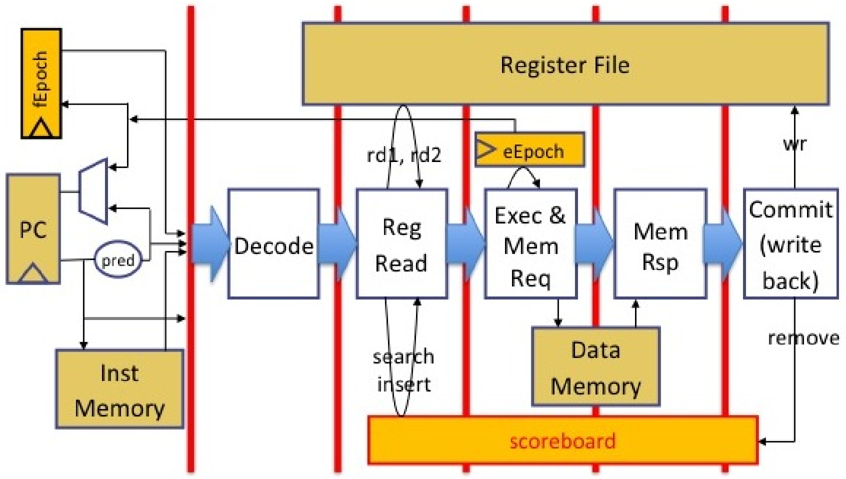


Figure 9.7: 6-stage pipeline

Fig. 9.7 illustrates a further refinement of our pipeline into 6 stages. The previous Fetch stage has been split into three stages: Fetch, Decode and RegRead (which performs the register reads). The previous Execute stage has been split into two stages: Execute ALU ops and Memory Request, and Memory Response. Data Memory accesses will take at least 1 cycle on a cache hit and may take many more cycles on a cache miss, so we always split it into a request action and a later response action.

9.5 Conclusion

The systematic application of pipelining requires a deep understanding of hazards, especially in the presence of concurrent operations. Performance issues are usually more subtle, compared to correctness issues.

In the next chapter we turn our attention back to control hazards, and we will look at the implementation of more sophisticated branch predictors that improve as execution proceeds by learning from the past branching of the program.

Chapter 10

Branch Prediction

10.1 Introduction

In this chapter, we turn our attention back to control hazards and study some increasingly sophisticated schemes for branch prediction. How important is this issue? Modern processor

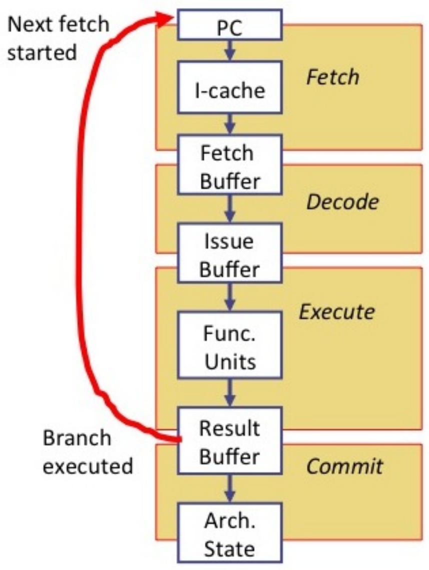


Figure 10.1: Loop penalty without branch prediction

pipelines are very deep; they may have ten or more stages. Consider the pipeline shown in Fig. 10.1. It is not until the end of the Execute stage, when a branch has been fully resolved, that we know definitively what the next PC should be. If the Fetch stage were always to wait until this moment to fetch the next instruction, we would effectively have no pipeline behavior at all. The time taken to execute a program loop containing N instructions would be $N\times$ *pipeline-depth*.

Of course, such a penalty might be acceptable if it were very rare. For example, if we only paid this penalty on actual branch instructions, and branch instructions were few and far between, then the penalty would be amortized over the high pipelining performance we would get on the other instructions.

Unfortunately this is not so in real programs. The SPEC benchmarks are a collection of programs intended to be representative of real application programs, and people have measured the statistics of instructions in these programs. The following table shows the dynamic instruction mix from SPEC, *i.e.,* we run the programs and count each kind of instruction actually executed.

Instruction type	SPECint92	SPECfp92
ALU	39%	13 %
FPU Add	—	20 %
FPU Mult	—	13 %
load	26 %	23 %
store	9 %	9 %
branch	16 %	8 %
other	10 %	12 %

SPECint92 programs:	compress, eqntott, espresso, gcc , li
SPECfp92 programs:	doduc, ear, hydro2d, mdijdp2, su2cor

In particular, the average *run length* between branch instructions is hardly 6 to 12 instructions, *i.e.,* branch instructions are quite frequent.

Our simple pipelined architecture so far performs very simple next-address predictions—it always predicts PC+4. This means that it always predicts that branches are not taken. This is not bad, since most instructions are not control-transfer instructions, and has been measured on real codes to have about a 70% accuracy.

In this chapter we will study improvements to this simple prediction scheme. Our improvements will be dynamic, *i.e.,* the prediction schemes "learn", or "self-tune" based on past program behavior to improve future prediction.

10.2 Static Branch Prediction

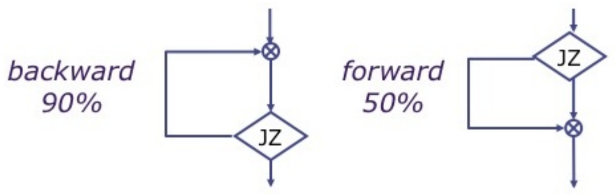


Figure 10.2: Branch probabilities for forward and backward branches

It turns out that in real codes, the probability that a branch is taken can correlate to the branch direction, *i.e.,* whether the branch is forward (towards a higher PC) or backward (towards a lower PC). Fig. 10.2 illustrates the situation.[1]

[1]Of course, these probabilities also depend on the compiler's code generation choices.

Based on this observation, an ISA designer can pro-actively attach preferred direction semantics to particular opcodes, so that compilers can take advantage of this. For example, in the Motorola MC88110, the `bne0` opcode (branch if not equal to zero) is a *preferred taken* opcode, *i.e.,* it is used in situations where the branch is more often taken. Conversely, the `beq0` opcode (branch if equal to zero) is a *preferred not taken* opcode, and is used in situations where the fall-through is more frequent.

The Hewlett Packard PA-RISC and the Intel IA-64 ISAs went further, allowing an arbitrary (static) choice of predicted direction. Measurements indicate that this achieves an 80% accuracy.

10.3 Dynamic Branch Prediction

Dynamic branch predictor have a "learning" or "training" component. As a program is executed, it records some information about how each branch instruction actually resolves (not taken or taken and, if taken, to which new PC). It uses this information in predictions for future executions of these branch instructions. If branch behavior were random, this would be useless. Fortunately, branch behaviors are often correlated:

- Temporal correlation: the way a branch resolves may be a good predictor of how it will resolve on the next execution. For example, the loop-exit branch for a loop with 1000 iterations will resolve the same way for the first 999 iterations. The return address of a function called in the loop may be the same for every iteration.
- Spatial correlation: Several branches may resolve in a highly correlated manner (e.g., to follow a preferred path of execution).

There are a large number of branch prediction schemes described in the literature and in actual products. Their importance grows with the depth of the processor pipeline, since the penalty is proportional to this depth. Even a particular processor implementation nowadays may use multiple branch prediction schemes. Each scheme is typically easy to understand in the abstract and in isolation. But there is considerable subtlety in how each scheme is integrated into the pipeline, and how multiple schemes might interact with each other. Fortunately, the atomic rule ordering semantics of BSV gives us a clean semantic framework in which to analyze and design correct implementations of such schemes. As illustrated in

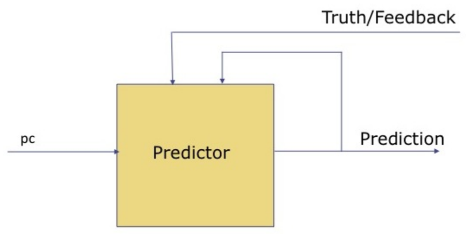


Figure 10.3: Dynamic prediction is a feedback control system

Fig. 10.3, dynamic prediction can be viewed as a dynamic feedback control system with the main mechanism sitting in the Fetch stage. It has two basic activities:

- *Predict*: In the forward direction, the Fetch logic needs an immediate prediction of the next PC to be fetched.
- *Update*: It receives feedback from downstream stages about the accuracy of its prediction ("truth"), which it uses to update its data structures to improve future predictions.

The key pieces of information of interest are: Is the current instruction a branch? Is it taken or not taken (direction)? If taken, what is the new target PC? These pieces of information may become known known at different stages in the pipeline. For example:

Instruction	Direction known after	Target known after
J	After Decode	After Decode
JR	After Decode	After Register Read
BEQZ, BNEZ	After Execute	After Decode

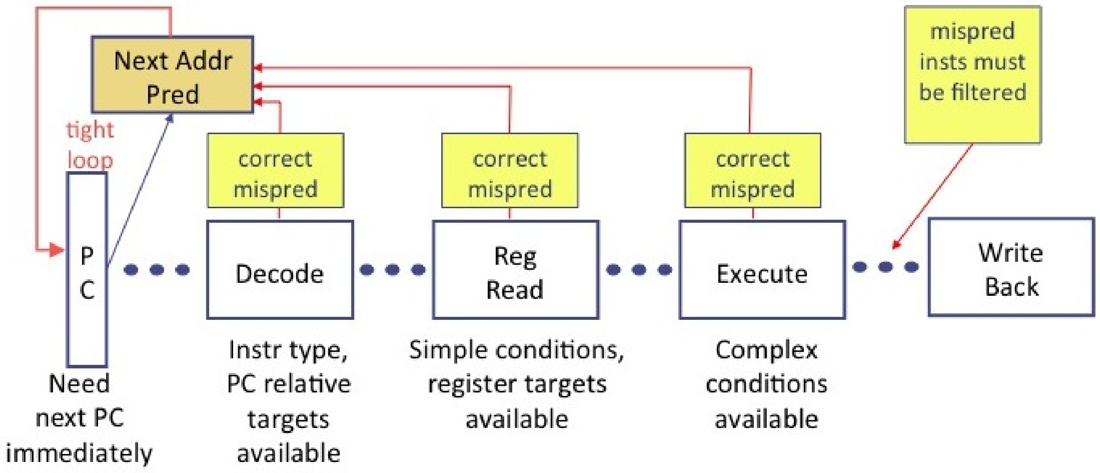

Next Addr Pred
tight loop
P C
correct mispred
Decode
correct mispred
Reg Read
correct mispred
Execute
mispred insts must be filtered
Write Back
Need next PC immediately
Instr type, PC relative targets available
Simple conditions, register targets available
Complex conditions available

Figure 10.4: Overview of control flow prediction

A predictor can redirect the PC only after it has the relevant information. Fig. 10.4 shows an overview of branch (control flow) prediction. In the Fetch stage there is a tight loop that requires a next-PC prediction for every instruction fetched. At this point the fetched instruction is just an opaque 32-bit value. After the Decode, we know the type of instruction (including opcodes). If it is an unconditional absolute or fixed PC-relative branch, we also know the target. After the Register Read stage, we know targets that are in registers, and we may also know some simple conditions. Finally, in the Execute stage, we have complete information. Of course, once we have recognized a misprediction, we must kill (filter out) all mispredicted instructions without them having any effect on architectural state.

Given a PC and its predicted PC (ppc), a misprediction can be corrected (used to redirect the pc) as soon as it is detected. In fact, pc can be redirected as soon as we have a "better" prediction. However, for forward progress it is important that a correct PC should never be redirected. For example, after the Decode stage, once we know that an instruction is a branch, we can use the past history of the that instruction to immediately check if the direction prediction was correct.

10.4 A first attempt at a better Next-Address Predictor (NAP)

Find another branch? There's a NAP for that!

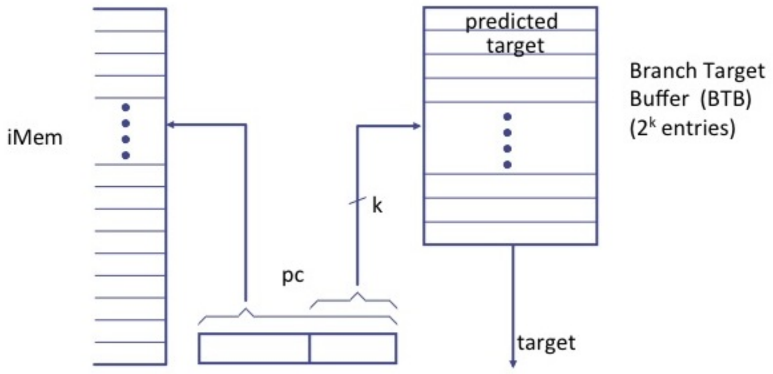


Figure 10.5: A first attempt at a better Next-Address Predictor

Fig. 10.5 illustrates a proposed Next-Address Predictor (NAP). Conceptually we could have a table mapping each PC in a program to another PC, which is its next-PC prediction. This table is called the *Branch Target Buffer* (BTB). Unfortunately, this table would be very large. So, we approximate it with a table that has just 2^k entries. For prediction, we simply fetch the next-PC from this table using k bits of the PC. Later, when we recognize a misprediction, we update this table with the corrected information.

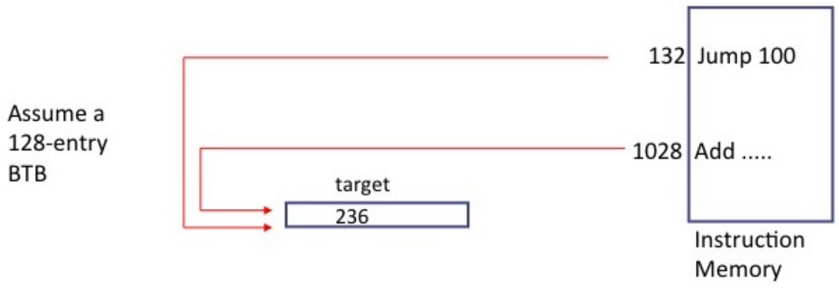


Figure 10.6: Address collisions in the simple BTB

Of course, the problem is that many instructions (where those k bits of their PCs are identical) will map into the same BTB location. This is called an address collision, or *aliasing* (a similar issue arises in caches, hash tables, etc.). Fig. 10.6 illustrates the problem. Instructions at PC 132 and 1028 have the same lower 7 bits, and so they will map to the same location in a 128-entry BTB. If we first encounter the Jump instruction at PC 132, the entry in the table will be 236 (132 + 4 + 100). Later, when we encounter the Add instruction at PC 1028, we will mistakenly predict a next-PC of 236, instead of 1032. Unfortunately, with such a simple BTB, this would be quite a common occurrence. Fortunately, we can improve our BTB to ameliorate this situation, which we discuss in the next section.

10.5 An improved BTB-based Next-Address Predictor

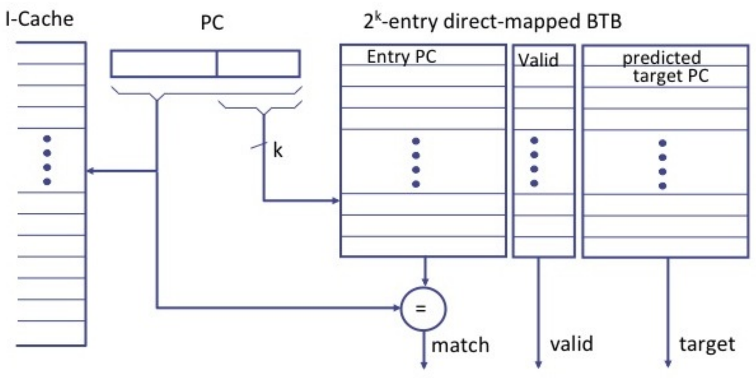


Figure 10.7: An improved BTB

This particular problem of address collisions is a familiar problem—we also encounter it in hash tables and caches. The solution is standard: if k bits are used for the table index, store the remaining $32 - k$ bits of the PC in the table as a *tag* that disambiguates multiple PCs that might map to the same location. This is illustrated in Fig. 10.7. Given a PC, we use k bits to index the table, and then we check the remaining $32 - k$ bits against the tag to verify that this entry is indeed for this PC and not for one of its aliases. If the entry does not match, we use a default prediction of PC+4.

Second, we only update the table with entries for actual branch instructions, relying on the default PC+4 prediction for the more common case where the instruction is not a branch. The “valid” bits in the table are initially False. When we update an entry we set its valid bit to True.

Note that this prediction is based only on the PC value of an instruction, *i.e.,* before we have decoded it. Measurements have shown that even very small BTBs, which now hold entries only for branch instructions, are very effective in practice.

10.5.1 Implementing the Next-Address Predictor

As usual, we first define the interface of our predictor:

Interface for next-address predictor

```
interface AddrPred;
   method Addr   predPc (Addr pc);
   method Action update (Redirect rd);
endinterface

typedef struct {
   Bool  taken;    // branch was taken
   Addr  pc;
   Addr  nextPC;
} Redirect
  deriving (Bits)
```

The `predPC` method is used in Fetch's tight loop to predict the next PC. The `update` method carries a `Redirect` struct that contains information about what really happened, to train the predictor. A simple PC+4 predictor module (which we've been using so far) implementing this interface is shown below:

Simple PC+4 predictor module

```
module mkPcPlus4(AddrPred);
   method Addr   predPc (Addr pc) = pc+4;
   method Action update (Redirect rd) = noAction;
endmodule
```

Next, we show the code for a BTB-based predictor. We use a register file for the actual table, which is fine for small tables, but we can easily substitute it with an SRAM for larger tables.

BTB-based predictor module

```
typedef struct {
   Addr    ppc;
   BtbTag  tag;
} BTBEntry
  deriving (Bits);

module mkBTB (AddrPred);
   RegFile #(BtbIndex, Maybe #(BTBEntry)) btb <- mkRegFileFull;

   function BtbIndex indexOf (Addr pc) = truncate (pc >> 2);
   function BtbTag   tagOf   (Addr pc) = truncateLSB (pc);

   method Addr predPc (Addr pc);
      BtbIndex index = indexOf (pc);
      BtbTag   tag   = tagOf (pc);
      if (btb.sub (index) matches tagged Valid .entry &&& entry.tag == tag)
         return entry.ppc;
      else
         return (pc + 4);
   endmethod

   method Action update (Redirect redirect);
      if (redirect.taken) begin
         let index = indexOfPC (redirect.pc);
         let tag   = tagOfPC   (redirect.pc);
         btb.upd (index,
                  tagged Valid (BTBEntry {Addr:redirect.nextPc, tag:tag}));
      end
   endmethod
endmodule
```

10.6 Incorporating the BTB-based predictor in the 2-stage pipeline

Below, we show our 2-stage pipeline (including scoreboard for data hazard management), modified to incorporate the BTB-based predictor. The changes are marked with *NEW* comments.

2-stage pipeline with BTB

```
module mkProc(Proc);
   Reg#(Addr)        pc <- mkRegU;
   RFile             rf <- mkRFile;
   IMemory         iMem <- mkIMemory;
   DMemory         dMem <- mkDMemory;
   Fifo#(Decode2Execute) d2e <- mkFifo;
   Reg#(Bool)    fEpoch <- mkReg(False);
   Reg#(Bool)    eEpoch <- mkReg(False);
   Fifo#(Addr) execRedirect <- mkFifo;
   AddrPred         btb <- mkBtb;                // NEW

   Scoreboard#(1) sb <- mkScoreboard;

   rule doFetch;
      let inst = iMem.req(pc);
      if (execRedirect.notEmpty) begin
         if (execRedirect.first.mispredict) begin    // NEW
            fEpoch <= !fEpoch;
            pc <= execRedirect.first.ppc;
         end
         btb.update (execRedirect.first);    // NEW
         execRedirect.deq;
      end
      else begin
         let ppc = btb.predPc (pc);    // NEW
         let dInst = decode(inst);
         let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
         if (! stall) begin
            let rVal1 = rf.rd1 (validRegValue (dInst.src1));
            let rVal2 = rf.rd2 (validRegValue (dInst.src2));
            d2e.enq (Decode2Execute {pc: pc,
                                     nextPC: ppc,
                                     dIinst: dInst,
                                     epoch: fEpoch,
                                     rVal1: rVal1,
                                     rVal2: rVal2});
            sb.insert(dInst.rDst); pc <= ppc; end
         end
      end
   endrule
```

```
41
42    rule doExecute;
43       let x = d2e.first;
44       let dInst = x.dInst; let pc    = x.pc;
45       let ppc   = x.ppc;   let epoch = x.epoch;
46       let rVal1 = x.rVal1; let rVal2 = x.rVal2;
47       if (epoch == eEpoch) begin
48          let eInst = exec(dInst, rVal1, rVal2, pc, ppc);
49          if (eInst.iType == Ld)
50             eInst.data <- dMem.req (MemReq{op:Ld,
51                                            addr:eInst.addr, data:?});
52          else if (eInst.iType == St) begin
53             let d <- dMem.req (MemReq{op:St,
54                                       addr:eInst.addr, data:eInst.data});
55          end
56          if (isValid (eInst.dst))
57             rf.wr (validRegValue(eInst.dst), eInst.data);
58          // ---- NEW begin
59          if (eInst.iType == J || eInst.iType == Jr || eInst.iType == Br)
60             execRedirect.enq (Redirect{pc: pc,
61                                        nextPc: eInst.addr,
62                                        taken: eInst.brTaken,
63                                        mispredict: eInst.mispredict,
64                                        brType: eInst.iType});
65          // ---- NEW end
66          if (eInst.mispredict) eEpoch <= !eEpoch;
67          d2e.deq;
68          sb.remove;
69    endrule
70 endmodule
```

In line 10 we instantiate the BTB. In the Fetch rule, in lines 17-20, we change the epoch and the PC only if `execRedirect` says it's describing a misprediction. In line 21 we update the BTB. In line 25 we use the BTB to perform our next-address prediction. In the Execute rule, in lines 58-65, if it is a branch-type instruction, we send information back to the BTB about what really happened.

10.7 Direction predictors

Conceptually, a direction predictor for a branch instruction is just a boolean (1 bit) that says whether the branch was taken or not. Unfortunately, using just 1 bit is fragile. Consider the branch for a loop-exit in a loop with 1000 iterations. For 999 iterations, it branches one way (stays in the loop), and the last iteration branches the other way (exits the loop). If we remember just 1 bit of information, we will only remember the exit direction, even though the probability is 0.999 in the other direction. This can be rectified by incorporating a little hysteresis into the control system. Suppose we remember 2 bits of information, treating it

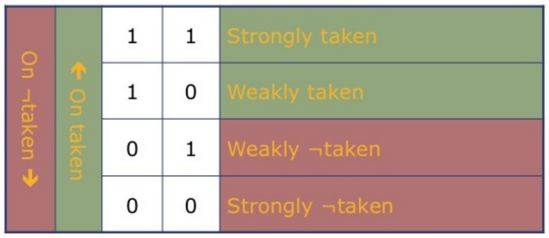


Figure 10.8: 2 bits to remember branch direction

as a 2-bit saturating counter. Every time we branch one way, we increment the counter, but we saturate (don't wrap around) at 3. When we branch the other way, we decrement the counter, but we saturate (don't wrap around) at 0. This is illustrated in Fig. 10.8. Now, it takes 2 mispredictions to change the direction prediction. Of course, this idea could be generalized to more bits, increasing the hysteresis.

Fig. 10.9 shows a Branch History Table to implement this idea. Note that this cannot be used any earlier than the Decode stage, because we consult the table only on Branch opcodes (which become known only in Decode). Again, we use k bits of the PC to index the table, and again we could use tags (the remaining $32 - k$ bits of the PC) to disambiguate due to aliasing. In practice, measurements show that even a 4K-entry BHT, with 2 bits per entry, can produce 80%-90% correct predictions.

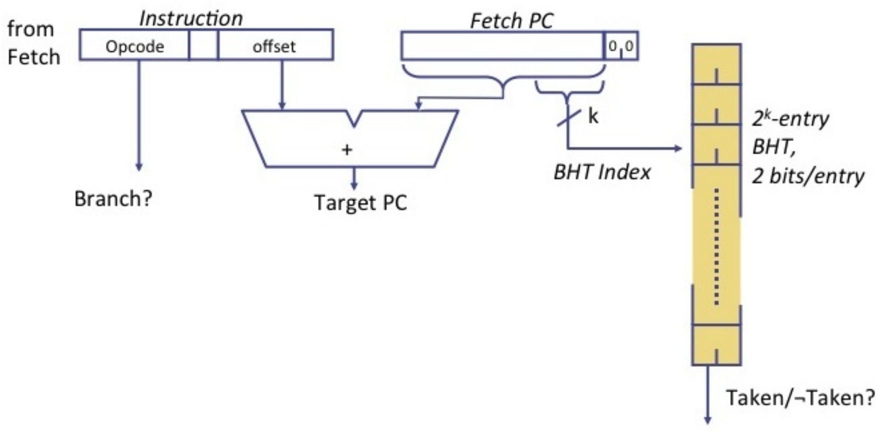


Figure 10.9: The Branch History Table (BHT)

Where does this information for updating the BHT come from? Once again, it comes from the Execute stage. In the next section, we discuss how multiple predictors, such as the BTB and BHT, are systematically incorporated into the pipeline.

10.8 Incorporating multiple predictors into the pipeline

Fig. 10.10 shows a sketch of an N-stage pipeline with just one prediction mechanism, the BTB (the figure only shows $N = 3$ stages, but the same structure holds for $N > 3$. In Execute, if there is an epoch mismatch, we mark the instruction as "poisoned" and send it on. Semantically, we are "dropping" or "discarding" the instruction, but in the implementation

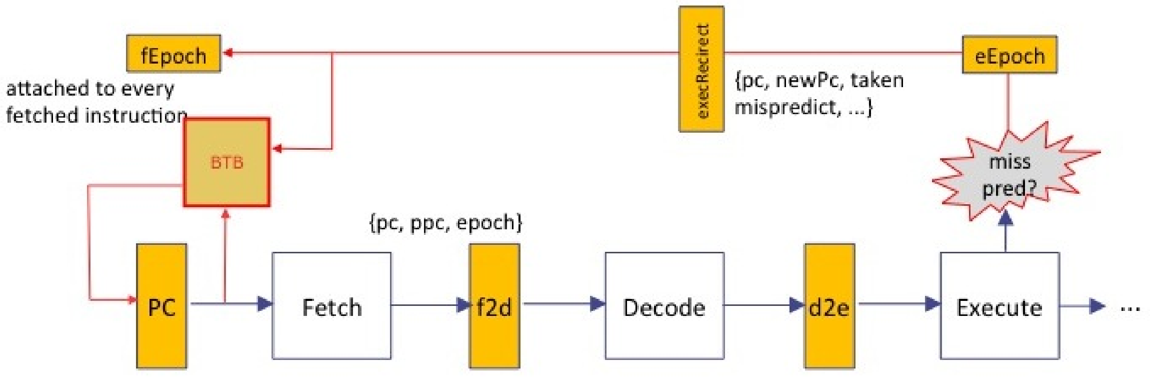


Figure 10.10: N-stage pipeline, with one predictor (BTB) only

the "poisoned" attribute tells downstream stages to use it only for book-keeping and cleanup, such as removing it from the scoreboard, and not to let it affect any architectural state. If there is no epoch mismatch but it is a branch instruction that was mispredicted, then change the `eEpoch`. If it is a branch, send the true information back to the Fetch stage.

In Fetch: when it receives a redirect message from Execute, use it to train the BTB, and to change epoch and redirect PC if necessary.

Fig. 10.11 shows how we incorporate 2 predictors into the pipeline, such as adding the BHT into the Decode stage. Since this stage, also, can now detect mispredictions, we provide it with its own epoch register `dEpoch` which is incremented when this happens. The Fetch stage now has two registers, `feEpoch` and `fdEpoch` which are approximations of Decode's `dEpoch` and Execute's `eEpoch`, respectively. Finally, Decode gets a `deEpoch` register that is an approximation of Execute's `eEpoch` register. In summary: each mispredict-detecting stage has its own epoch register, and each stage has a register that approximates downstream epoch registers. In all this, remember that Execute's redirect information (the truth) should never be overridden.

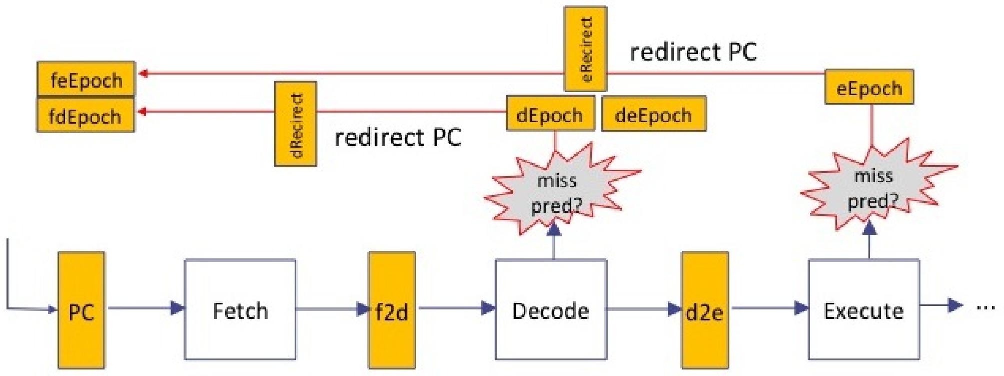


Figure 10.11: N-stage pipeline, with two predictors

Fig. 10.12 illustrates how redirection works. The Fetch stage now attaches two epoch numbers to each instruction it sends downstream, the values from both `fdEpoch` and `feEpoch`. We refer to these values with the instruction as `idEp` and `ieEp`, respectively.

The Execute stage behaves as before, comparing the `ieEp` (on the instruction) with

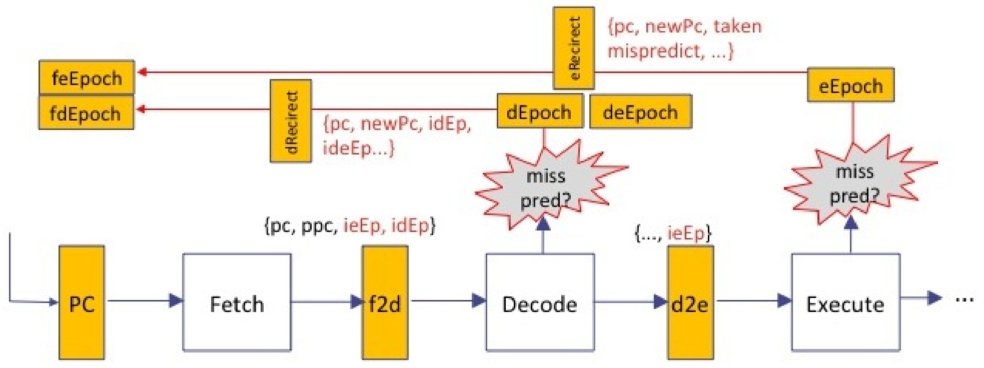


Figure 10.12: N-stage pipeline, with two predictors and redirection logic

its `eEpoch` register. In Decode, if `ieEp` (on the instruction) is different from its `deEpoch` register, then it now learns that the Execute stage has redirected the PC. In this case, it updates both its epoch registers `dEpoch` and `deEpoch` from the values on the instruction, `idEp` and `ieEp`, respectively. Otherwise, if `idEp` (on the instruction) is different from its `dEpoch` register, then this is a wrong-path instruction with respect to on one of its own redirections, and so it drops the instruction. For non-dropped instructions, if the predicted PC on the instruction differs from its own prediction (using the BHT), it updates its `dEpoch` register and sends the new information to the Fetch stage.

The Fetch stage can now receive redirect messages from both the Execute and Decode stages. Messages from Execute are treated as before. For a message from Decode, if `ideEp` $\neq$ `feEpoch`, then this is a message about a wrong-path instruction, so we ignore this message. Otherwise, we respond to the message by redirecting the PC and updating `fdEpoch`.

10.8.1 Extending our BSV pipeline code with multiple predictors

Now that we understand the architectural ideas behind incorporating multiple predictors in the pipeline, let us modify our BSV code for the pipeline accordingly. We shall work with a 4-stage pipeline, F (Fetch), D&R (Decode and Register Read), E&M (Execute and Memory), and W (Writeback). This version will have no predictor training, so messages are sent only for redirection.

2-stage pipeline with BTB

```
module mkProc (Proc);
   Reg#(Addr)         pc <- mkRegU;
   RFile              rf <- mkBypassRFile;
   IMemory          iMem <- mkIMemory;
   DMemory          dMem <- mkDMemory;
   Fifo#(1, Decode2Execute) d2e <- mkPipelineFifo;
   Fifo#(1, Exec2Commit)    e2c <- mkPipelineFifo;
   Scoreboard#(2) sb <- mkPipelineScoreboard;
   // ---- NEW section begin
   Reg#(Bool)    feEp <- mkReg(False);
   Reg#(Bool)    fdEp <- mkReg(False);
   Reg#(Bool)    dEp <- mkReg(False);
   Reg#(Bool)    deEp <- mkReg(False);
   Reg#(Bool)    eEp <- mkReg(False);
   Fifo#(ExecRedirect) execRedirect <- mkBypassFifo;
   Fifo#(DecRedirect) decRedirect <- mkBypassFifo;
   AddrPred#(16) addrPred <- mkBTB;
   DirPred#(1024) dirPred <- mkBHT;
   // ---- NEW section end

   rule doFetch;
      let inst = iMem.req(pc);
      if (execRedirect.notEmpty) begin
         feEp <= !feEp;
         pc <= execRedirect.first.newPc;
         execRedirect.deq;
      end
      else if (decRedirect.notEmpty) begin
         if (decRedirect.first.eEp == feEp) begin
            fdEp <= ! fdEp;
            pc <= decRedirect.first.newPc;
         end
         decRedirect.deq;
      end
      else begin
         let ppc = addrPred.predPc (pc);                    // NEW
         f2d.enq (Fetch2Decoode {pc: pc, ppc: ppc, inst: inst,
                                eEp: feEp, dEp: fdEp});    // NEW
      end
   endrule

   function Action decAndRegFetch(DInst dInst, Addr pc, Addr ppc, Bool eEp);
      action
         let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2)
                     || sb.search3(dInst.dst);    // WAW hazard check
         if (!stall) begin
            let rVal1 = rf.rd1 (validRegValue (dInst.src1));
            let rVal2 = rf.rd2 (validRegValue (dInst.src2));
            d2e.enq (Decode2Execute {pc: pc, ppc: ppc,
                                     dInst: dInst, epoch: eEp,
                                     rVal1: rVal1, rVal2: rVal2});
            sb.insert (dInst.rDst);
         end
      endaction
   endfunction

   rule doDecode;
      let x = f2d.first; let inst = x.inst; let pc = x.pc;
      let ppc = x.ppc; let idEp = x.dEp; let ieEp = x.eEp;
      let dInst = decode(inst);
      let newPc = dirPrec.predAddr(pc, dInst);
      if (ieEp != deEp) begin // change Decode's epochs and
                              // continue normal instruction execution
         deEp <= ieEp; let newdEp = idEp;
         decAndRegRead (inst, pc, newPc, ieEp);
         if (ppc != newPc) begin
            newDEp = !newdEp;
            decRedirect.enq (DecRedirect {pc: pc,
                                          newPc: newPc, eEp: ieEp});
         end
         dEp <= newdEp
      end
      else if (idEp == dEp) begin
         decAndRegRead (inst, pc, newPc, ieEp);
         if (ppc != newPc) begin
            dEp <= !dEp;
            decRedirect.enq (DecRedirect {pc: pc,
                                          newPc: newPc, eEp: ieEp});
         end
      end
      f2d.deq;
   endrule

   rule doExecute;
      let x = d2e.first;
      let dInst = x.dInst; let pc    = x.pc;
      let ppc   = x.ppc;   let epoch = x.epoch;
      let rVal1 = x.rVal1; let rVal2 = x.rVal2;
      if (epoch == eEpoch) begin
         let eInst = exec (dInst, rVal1, rVal2, pc, ppc);
         if (eInst.iType == Ld)
            eInst.data <- dMem.req (MemReq {op:Ld, addr:eInst.addr, data:?});
         else if (eInst.iType == St)
            let d <- dMem.req (MemReq {op:St, addr:eInst.addr, data:eInst.data});
         e2c.enq (Exec2Commit {dst:eInst.dst, data:eInst.data});
         if (eInst.mispredict) begin
            execRedirect.enq (eInst.addr);
            eEpoch <= !eEpoch;
         end
      end
      else
         e2c.enq (Exec2Commit {dst:Invalid, data:?});
      d2e.deq;
   endrule

   rule doCommit;
      let dst  = eInst.first.dst;
      let data = eInst.first.data;
      if (isValid (dst))
         rf.wr (validValue(dst), data);
      e2c.deq;
      sb.remove;
   endrule
```

Lines 10-19 instantiate all the new modules and state needed for the predictors. Line 38, in the Fetch rule, attaches two epoch values to each instruction sent downstream. Lines 62-80 in the Decode rule perform the decode-stage prediction that we just described. They use the function shown on lines 42-55. The Execute and Commit rules are completely unchanged from before.

10.9 Conclusion

This chapter has again shown the value of thinking about processor pipelines as distributed systems (logically independent units communicating with messages) instead of as a globally synchronous state machine. Trying to manage the complex concurrency issues around multiple branch predictors by thinking about the pipeline as a globally synchronous machine can be very difficult and error-prone. The concept of epochs, distributed approximations of the epoch, and messages whose exact latency does not affect correctness, allow us to design a solution that is clean and modular, and whose correctness is easier to reason about.

Chapter 11

Exceptions

11.1 Introduction

Interrupts, traps and exceptions are “infrequent” events that occur unpredictably during normal instruction execution. These could be *synchronous* or *asynchronous*. Synchronous exceptions occur due to some problem during a particular instruction’s execution: undefined opcodes, privileged instructions attempted in user mode, arithmetic overflows, divide by zero, misaligned memory accesses, page faults, TLB misses, memory access protection violations, traps into the OS kernel, and so on. Asynchronous exceptions occur due to some external event requiring the processor’s attention: timer expiry, signal for an event of interest in an I/O device, hardware failures, power failures, and so on.

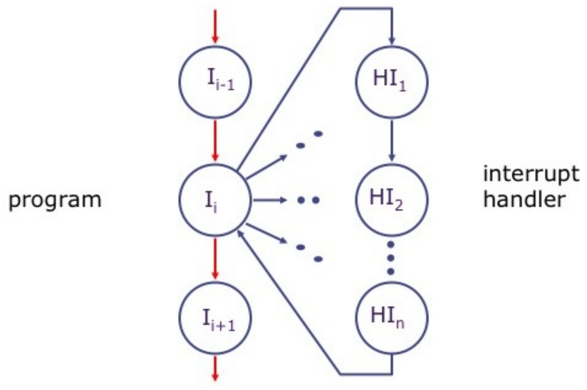


Figure 11.1: Instruction flow on an interrupt

When such an interrupt occurs, the normal flow of instructions is temporarily suspended, and the processor executes code from a *handler*. This is illustrated in Fig. 11.1. When the handler completes, the processor resumes the interrupted normal flow. Of course, we must take care that handler execution does not disturb the architectural state of the normal flow, so that it can be resumed cleanly.

11.2 Asynchronous Interrupts

An asynchronous interrupt is typically initiated by an external device (such as a timer or I/O device) by asserting one of the *prioritized interrupt request lines* (IRQs) to get the processor's attention. The processor may not respond immediately, since it may be in the middle of a critical task (perhaps even in the middle of responding to a previous interrupt).

Most ISAs have a concept of *precise interrupts*, *i.e.,* even though the processor may have a deep pipeline with many instructions simultaneously in flight, even though it may be executing instructions in superscalar mode or out of order, there is always a well-defined notion of the interrupt occurring precisely between two instructions I_{j-1} and I_j such that it is as if I_{j-1} has completed and updated its architectural state, and I_j has not even started.

When the interrupt occurs, the processor saves the PC of instruction I_j in a special register, `epc`. It disables further interrupts, and jumps to a designated interrupt handler and starts running in kernel mode. Disabling is typically done by writing to an *interrupt mask register* which has a bit corresponding to each kind of interrupt or each external interrupt line; when a bit here is 0, the processor ignores interrupts from that source. Kernel mode (also sometimes called supervisor mode or privileged mode) is a special mode allowing the processor full access to all resources on the machine; this is in contrast to user mode, which is the mode in which normal application code is executed, where such accesses are prohibited (by trapping if they are attempted). Most ISAs have this kernel/user mode distinction for protection purposes, *i.e.,* it prevents ordinary application programs from accessing or damaging system devices or other application programs that may be time multiplexed concurrently by the operating sytem.

At the speeds at which modern processors operate, interrupts are a major disruption to pipelined execution, and so raw speed of interrupt handling is typically not of paramount concern.

11.2.1 Interrupt Handlers

In effect, the processor is redirected to a handler when an interrupt occurs. What is the PC target for this redirection? In some machines it is a known address fixed by the ISA. Other machines have an *interrupt vector* at a known address, which is an array of redirection PCs. When an interrupt occurs, the kind and source of the interrupt indicate an index into this array, from which the processor picks up the redirection PC.

Once the PC has been redirected and the handler starts executing, it saves the `epc` (the PC of the instruction to resume on handler completion). For this, the ISA has an instruction to move the `epc` into a general purpose register. At least this much work must be done before the handler re-enables interrupts in order to handle nested interrupts. The ISA typically also has a `status` register that indicates the cause or source of the interrupt, so that the handler can respond accordingly.

On handler completion, it executes a special indirect jump instruction ERET (return from exception) which:

- enables interrupts,
- restores the processor to user mode from kernel mode, and

- restores the hardware status and control state so that instruction resumes where it left off at the interrupted instruction.

11.3 Synchronous Interrupts

A synchronous interrupt is caused by a particular instruction, and behaves like a control hazard, *i.e.,* the PC has to be redirected and instructions that follow in the normal flow that are already in the pipeline have to be dropped, just like wrong-path instructions after a misprediction. As mentioned before, the ISA defines a special register `epc` in which the processor stores PC+4 of the interrupted instruction, and a special ERET instruction to return to normal operation after an exception handler.

Synchronous interrupts come in two flavors:

- *Exceptions*: The instruction cannot be completed (e.g., due to a page fault) and needs to be *restarted* after the exception has been handled.
- *Faults* or *Traps*: These are like system calls, *i.e.,* deliberate calls into kernel mode routines, and the instruction is regarded as completed.

11.3.1 Using synchronous exceptions to handle complex and infrequent instructions

Synchronous exceptions offer an alternative implementation for complex or infrequent instructions. For example, some processors do not implement floating point operations in hardware, because of their complexity in hardware. In such an implementation, an instruction like:

```
mult ra, rb
```

causes an exception, and the handler implements a floating point multiplication algorithm in software. When the instruction is encountered, PC+4 is stored in `epc`, the handler performs the multiplication and returns using ERET, and execution continues at PC+4 as usual.

11.3.2 Incorporating exception handling into our single-cycle processor

We need to extend many of our interfaces to express exception-related information wherever necessary, in addition to the normal information. For example, a memory request will now return a 2-tuple, {memory response, memory exception}. Our instruction definitions are extended for instructions like `eret` and `mult`, and so on.

New instructions for exceptions
```
typedef enum {Unsupported, Alu, Ld, St, J, Jr, Br, Mult, ERet}
        IType
    deriving (Bits, Eq);
```

Decode has to be extended to handle the new opcodes:

Decode

```
Bit#(6) fcMULT  = 6'b011000;    // mult opcode
Bit#(5) rsERET  = 5'b10000;     // eret opcode

function DecodedInst decode(Data Inst);
   DecodedInst dInst = ?;
   ...
   case
      ...
      opFUNC: begin
                 case (funct)
                    ...
                    fcMULT:
                       dInst.iType  = Mult;
                       dInst.brFunc = AT;
                       dInst.rDst   = Invalid;
                       dInst.rSrc1  = validReg (rs);
                       dInst.rSrc2  = validReg (rt);
              end
      opRS: begin
               ...
               if (rs==rsERET) begin
                  dInst.iType  = ERet;
                  dInst.brFunc = AT;
                  dInst.rDst   = Invalid;
                  dInst.rSrc1  = Invalid;
                  dInst.rSrc2  = Invalid;
               end
            end
   endcase
   return dInst;
endfunction
```

We modify the code for branch address calculation to hande `ERet` and to redirect `Mult` to its software handler:

Branch address calculation

```
function Addr brAddrCalc (Addr pc, Data val,
                          IType iType, Data imm, Bool taken, Addr epc);
   Addr pcPlus4 = pc + 4;
   Addr targetAddr =
      case (iType)
         J  : pcPlus4[31:28], imm[27:0];
         Jr : val;
         Br : (taken? pcPlus4 + imm : pcPlus4);
        Mult: h'1010;    // Address of multiplication handler
        ERet: epc;
        Alu, Ld, St, Unsupported: pcPlus4;
      endcase;
   return targetAddr;
endfunction
```

In the code for instruction execution, we pass `epc` as an additional argument to the branch address calculator:

Instruction execution

```
function ExecInst exec (DecodedInst dInst, Data rVal1,
                        Data rVal2, Addr pc, Addr epc);
   ...
   let brAddr = brAddrCalc (pc, rVal1, dInst.iType,
                            validValue(dInst.imm), brTaken, epc );  ...
   eInst.brAddr = ... brAddr ...;
   ...
   return eInst;
endfunction
```

In the Execute rule, we load `epc` if it is mult instruction:

Execute rule

```
   rule doExecute;
      let inst = iMem.req (pc);
      let dInst = decode (inst);
      let rVal1 = rf.rd1 (validRegValue (dInst.src1));
      let rVal2 = rf.rd2 (validRegValue (dInst.src2));
      let eInst = exec (dInst, rVal1, rVal2, pc, epc);
      if (eInst.iType == Ld)
         eInst.data <- dMem.req (MemReq {op: Ld, addr:
                                         eInst.addr, data: ?});
      else if (eInst.iType == St)
         let d <- dMem.req (MemReq {op: St, addr:
                                   eInst.addr, data: eInst.data});
      if (isValid (eInst.dst))
         rf.wr (validRegValue (eInst.dst), eInst.data);
      pc <= eInst.brTaken ? eInst.addr : pc + 4;
      if (eInst.iType == Mult) epc <= eInst.addr;
   endrule
```

With these changes, our single-cycle implementation handles exceptions.

11.4 Incorporating exception handling into our pipelined processor

Fig. 11.2 shows that synchronous exceptions can be raised at various points in the pipeline. In Fetch, the PC address may be invalid or we may encounter a page fault. In Decode, the

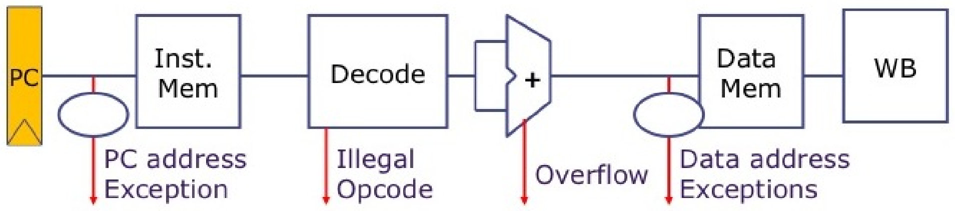


Figure 11.2: Synchronous interrupts raised at various points in the pipeline

opcode may be illegal or unimplemented. In Execute, there may be an overflow, or a divide by zero. In Memory, the data address may be invalid or we may encounter a page fault.

In fact, a single instruction can raise multiple exceptions as it passes through the pipeline. If so, which one(s) should we handle? This will be specified in the ISA (which makes no mention of the implementation question of pipelining). In an ISA, an instruction's functionality is often described in pseudocode, and this will provide a precise specifcation of which potential exceptions should be handled, and in what order. Normally, this order will correspond to the upstream-to-downstream order in implementation pipelines.

In the pipeline, if multiple instructions raise exceptions, which one(s) should we process? This is an easier question, since the ISA is specified with one instruction-at-a-time execution: clearly we should handle the exception from the oldest instruction in the pipeline.

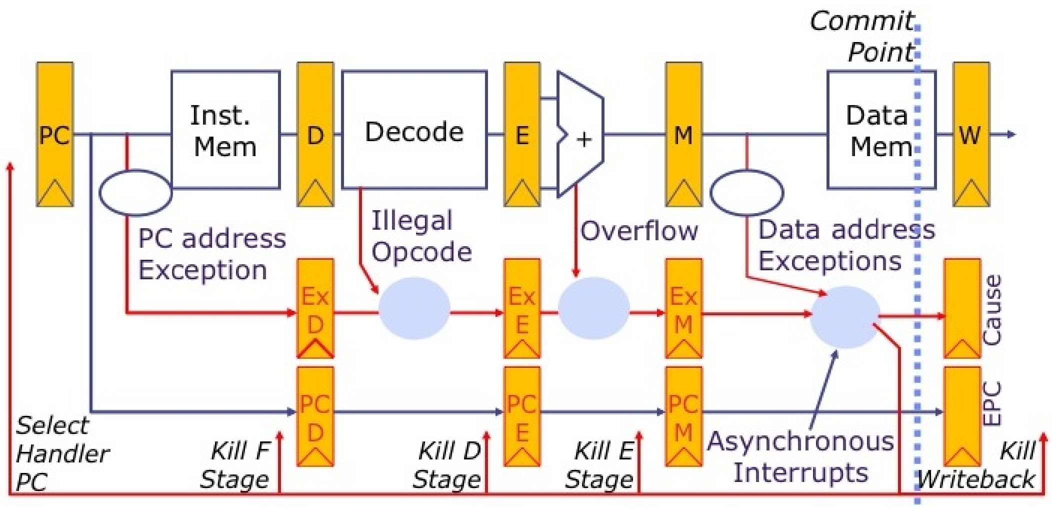


Figure 11.3: Exception handling in the processor pipeline

Fig. 11.3 illustrates that, in addition to the normal values carried down the pipeline (top of diagram), we must also carry along exception information and the PC of the instruction causing the exception (bottom of diagram). Further, we must have the ability to kill younger instructions.

In order to implement the idea of precise exceptions, we do not "immediately" respond to an exception; we carry the information along with the instruction until the final Commit stage, where instruction ordering is finally resolved. Remember also that the exception may be raised by an instruction that is later discovered to be a wrong-path instruction, in which case we should ignore it as part of killing that instruction; these questions are resolved when we reach the final Commit stage. Similarly, we can also inject external interrupts in the Commit stage, where we have precise knowledge of actual instruction ordering; in this case, we kill all instructions that logically follow the injection point.

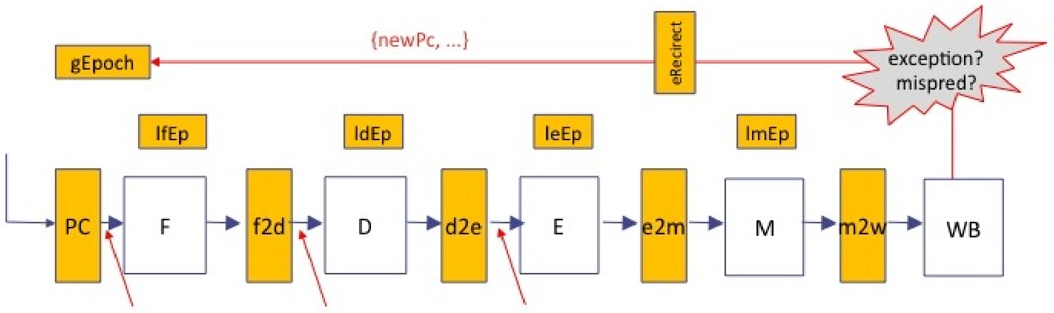


Figure 11.4: Redirection logic in the processor pipeline

Other than these exception-specific features, an exception raised in stage j is just like discovering a misprediction in stage j: we use the same epoch mechanisms and kill/poison mechanisms that we saw in dealing with control hazards in Chapter 6 to deal with redirection and instruction-dropping. This is sketched in Fig. 11.4. Every instruction carries along three pieces of information (amongst other things):

- epoch: the global epoch number with which it entered the pipeline
- poisoned: a boolean indicating whether a previous stage has already declared that this instruction should be dropped
- cause: if a previous stage raised an exception, this contains the cause

Each stage contains a local epoch register `stageEp`.

Each stage pseudocode

```
if (poisoned || epoch < stageEp)
      pass through <epoch:epoch, poisoned:True, cause:?, ...>
else
   if (cause == None)
      local_cause <- do stage work
      pass <epoch:epoch, poisoned:False, cause:local_cause, ...>
   else
      local_cause = False;
      pass <epoch:epoch, poisoned:False, cause:cause, ...>
stageEp <= ((local_cause != None) ? (epoch+1) : epoch);
```

If an exception makes it successfully into the Commit stage, then we store the information in the `cause` and `epc` registers, and redirect Fetch to the exception handler.

Commit stage pseudocode

```
if (! poisoned)
   if ((cause != None) || mispredict)
      redirect.enq (redirected PC)
```

Let us revisit the question of the size (bit-width) of the epoch values. We need to distinguish 3 cases:

- <: represents a wrong-path instruction
- =: represents a normal instruction
- >: represents a normal path instruction where a later stage caused a redirect

Thus, 3 values are enough, so 2 bits are enough.

11.4.1 BSV code for pipeline with exception handling

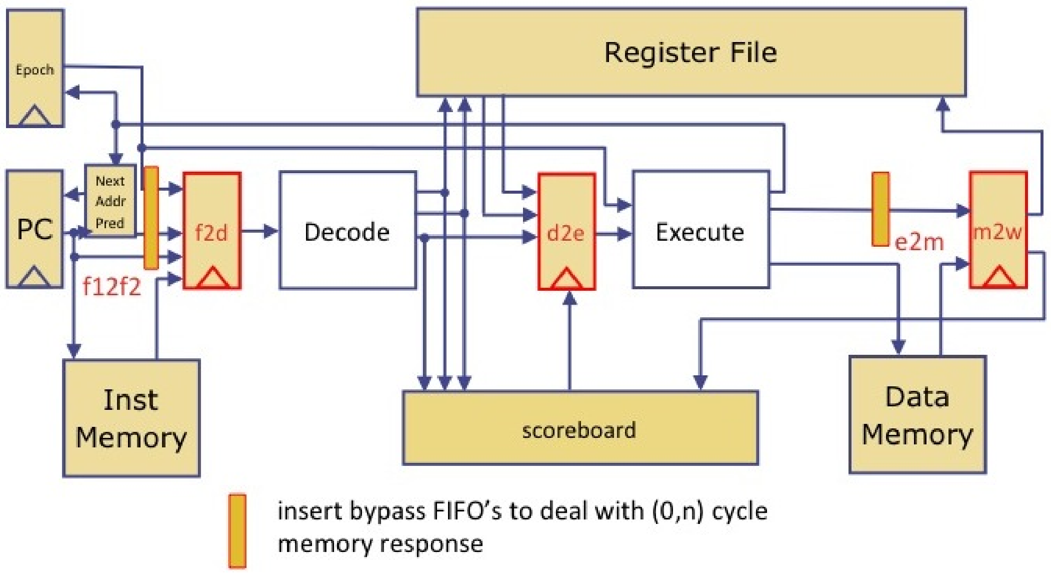


Figure 11.5: 4 stage pipeline in which we add exception handling

In the rest of this chapter we present code for a pipeline with exception handling. As a baseline, we take the 4-stage pipeline illustrated in Fig. 11.5: Fetch (F), Decode and Register Read (D&R), Execute and Memory (E&M), and Writeback (W). It has a single branch predictor and no training, *i.e.,* messages are only sent for redirection. We also integrate Instruction and Data memories. Because (for the moment) we take these to have zero-latency (combinational) reads, we balance the corresponding part of the pipeline with BypassFIFOs as shown in the figure.

Pipeline with exception handling

```
module mkProc(Proc);
   Reg #(Addr) pc <- mkRegU;
   Rfile rf <- mkBypassRFile;
   Cache #(ICacheSize) iCache <- mkCache;
   Cache #(DCacheSize) dCache <- mkCache;

   FIFO #(1, Fetch2Decode) f12f2 <- mkBypassFifo;
   FIFO #(1, EInst) e2m <- mkBypassFifo;
   FIFO #(1, Decode2Execute) d2e <- mkPipelineFifo;
   FIFO #(1, Exec2Commit)    e2c <- mkPipelineFifo;

   Scoreboard #(2) sb <- mkPipelineScoreboard;

   Reg #(Epoch) globalEp <- mkReg(False);
   FIFO #(ExecRedirect) execRedirect <- mkBypassFifo;
   AddrPred #(16) addrPred <- mkBTB;

   Reg #(Epoch)    fEp <- mkReg(False);  // Present in 2nd fetch stage
   Reg #(Epoch)    dEp <- mkReg(False);
   Reg #(Epoch)    eEp <- mkReg(False);
   Reg #(Epoch)    mEp <- mkReg(False);  // Present in 2nd execute stage

   rule doFetch1;
      if (execRedirect.notEmpty) begin
         globalEp <= next(globalEp);
         pc <= execRedirect.first.newPc;
         execRedirect.deq;
      end
      else begin
         iCache.req (MemReq {op: Ld, addr: pc, data:?});
         let ppc = addrPred.predPc (pc);
         f12f2.enq (Fetch2Decode {pc: pc, ppc: ppc, inst: ?,
                                  cause: None, epoch: globalEp});
      end
   endrule

   rule doFetch2;
      match inst, mCause <- iCache.resp;
      f12f2.deq;
      let f2dVal = f12f2.first;
      if (lessThan (f2dVal.epoch, fEp)) begin
         /* discard */
      end
      else begin
         f2dVal.inst = inst;
        f2dVal.cause = mCause;
        f2d.enq (f2dVal);
        if (mCause != None)
           fEp <= next (fEp);
      end
   endrule

   rule doDecode;
      let x = f2d.first;
      let inst = x.inst;
      let cause = x.cause;
      let pc = x.pc;
      let ppc = x.ppc;
      let epoch = x.epoch;
      if (lessThan (epoch, dEp)) begin
         /* discard */
      end
      else
         if (cause != None) begin
            d2e.enq (Decode2Execute {pc: pc, ppc: ppc,
                                     dInst: ?, cause: cause, epoch: epoch,
                                     rVal1: ?, rVal2: ?});
            dEp <= epoch;
         end
      else begin
         //  Not poisoned and no exception in the previous stage
         match dInst, dCause = decode(inst);
         let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2)
                        || sb.search3(dInst.dst);
         if (dCause != None) begin
            d2e.enq (Decode2Execute {pc: pc, ppc: ppc,
                                     dInst: ?, cause: dCause,
                                     epoch: epoch,
                                     rVal1: ?, rVal2: ?});
            dEp <= next(epoch);
         end
         else if (!stall) begin
            let rVal1 = rf.rd1 (validRegValue(dInst.src1));
            let rVal2 = rf.rd2 (validRegValue(dInst.src2));
            d2e.enq (Decode2Execute {pc: pc, ppc: ppc,
                                     dInst: dInst, cause: dCause,
                                     epoch: epoch,
                                     rVal1: rVal1, rVal2: rVal2});
            dEp <= epoch;
            sb.insert(dInst.rDst);
         end
      end
      f2d.deq;
   endrule

   //  pc redirect has been moved to the Commit stage where exceptions are resolved
   rule doExecute1;
      let x = d2e.first;
      let dInst = x.dInst;
      let pc = x.pc;
      let ppc = x.ppc;
      let epoch = x.epoch;
      let rVal1 = x.rVal1;
      let rVal2 = x.rVal2;
      let cause = x.cause;
      if (lessThan (epoch, eEpoch))
```

```
         e2m.enq (EInst {pc: pc, poisoned: True});
      else begin
         if (cause != None) begin
            eEp <= epoch;
            e2m.enq (EInst {pc: pc, poisoned: False, cause: cause});
         end
         else begin
            let eInst = exec (dInst, rVal1, rVal2, pc, ppc);
            e2m.enq (eInst);
            if (eInst.cause == None) begin

               //  Memory operations should be done only for non-exceptions
               //  and non-poisoned instructions (next slide)
               if (eInst.iType == Ld)
                  dCache.req (MemReq {op:Ld, addr:eInst.addr, data:? });
               else if (eInst.iType == St)
                  dCache.req (MemReq {op:St,addr:eInst.addr,
                                      data:eInst.data});

               if (eInst.mispredict)
                  eEp <= next(epoch);
               else
                  eEp <= epoch;
            end
            else
               eEp <= next(epoch);
         end
      end
      d2e.deq;
   endrule

   //  Data Memory response stage
   rule doExecute2;
      let x = e2m.first;
      let eInst = x.eInst;
      let poisoned = x.poisoned;
      let cause = x.cause;
      let pc = x.pc;
      let epoch = x.epoch;
      if (poisoned || lessThan (epoch,mEp)) begin
         eInst.poisoned = True;
         e2c.enq (Exec2Commit {eInst: eInst, cause: ?, pc: ?, epoch: ?});
      end
      else begin
         if (cause != None) begin
            mEp <= epoch;
            e2c.enq (Exec2Commit {eInst:eInst,cause:cause,
                                  pc:pc,epoch:epoch});
         end
      else begin
         if (eInst.iType == Ld || eUbst.iType == St) begin
            match data, mCause <- dCache.resp;  eInst.data = data;
            if (mCause == None)
               mEp <= epoch;
            else
               mEp <= next(epoch);
            e2c.enq (Exec2Commit {eInst:eInst, cause:mCause, pc:pc});
         end
         else begin
            mEp <= epoch;
            e2c.enq (Exec2Commit {eInst:eInst, cause:None, pc:pc});
         end
      end
   endrule

   rule doCommit;
      let x = e2c.first;
      let cause = x.cause;
      let pc = x.pc;
      e2c.deq;
      if (! (iMemExcep (cause) || iDecodeExcep (cause)))
         sb.remove;
      if (! x.eInst.poisoned) begin
         if (cause == None) begin
            let y = validValue (x.eInst);
            let dst = y.dst;
            let data = y.data;
            if (isValid (dst))
               rf.wr (tuple2 (validValue (dst), data);
            if (eInst.mispredict)
               execRedirect.enq(eInst.addr);
         end
         else begin
            let newpc = excepHandler (cause);
            eret <= pc;
            execRedirect.enq (newpc);
         end
      end
   endrule
endmodule
```

Chapter 12

Caches

12.1 Introduction

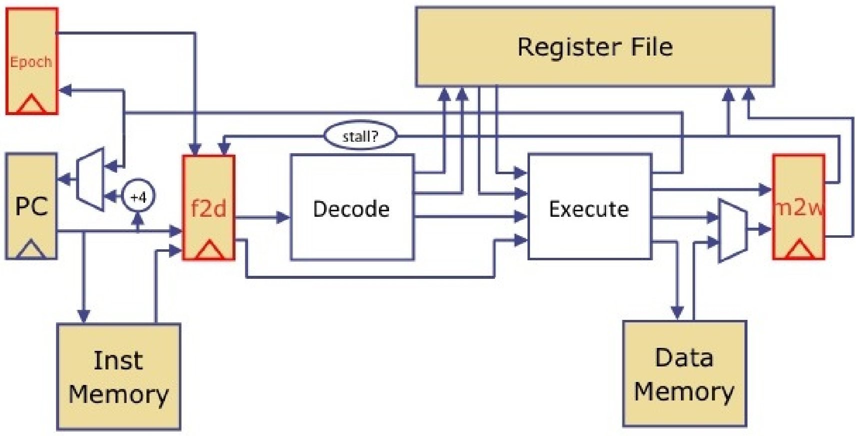


Figure 12.1: 4 stage pipeline

Fig. 12.1 recaps our 4-stage pipeline structure from previous chapters. So far, we have just assumed the existence of an Instruction Memory and a Data Memory which we can access with zero delay (combinationally). Such “magic” memories were useful for getting started, but they are not realistic.

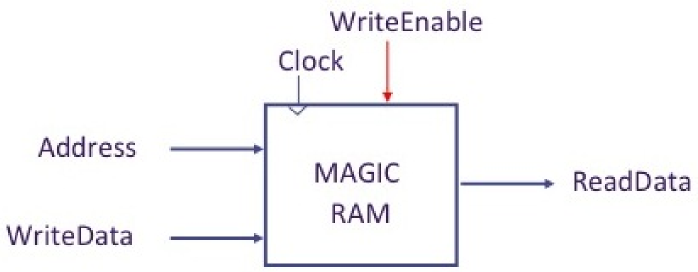


Figure 12.2: Simple memory model

Fig. 12.2 shows a model of one of these simple memories. When an address is presented to the module, the data from the corresponding location is available immediately on the ReadData bus. If WriteEnable is asserted, then the WriteData value is immediately written to the corresponding address. In real systems, with memories of realistic sizes and a realistic "distance" from the processor, reads and writes can take many cycles (even 100s to 1000s of cycles) to complete.

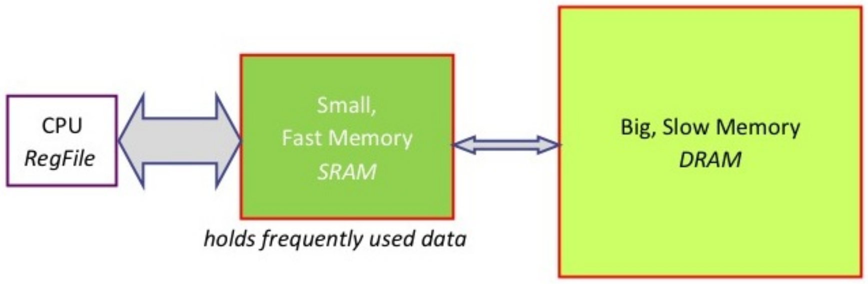


Figure 12.3: A memory hierarchy

Unfortunately it is infeasible to build a memory that is simultaneously large (capacity), fast (access time) and consumes low power. As a result, memories are typically organized into hierarchies, as illustrated in Fig. 12.3. The memory "near" the processor is typically an SRAM (Static Random Access Memory), with limited capacity and high speed. This is backed up by a much larger (and slower) memory, usually in DRAM (Dynamic RAM) techology. The following table gives a general comparison of of their properties (where << means "much less than"):

size (capacity in bytes):	RegFile	<<	SRAM	<<	DRAM
density (bits/mm^2):	RegFile	<<	SRAM	<<	DRAM
latency (read access time):	RegFile	<<	SRAM	<<	DRAM
bandwidth (bytes per second):	RegFile	>>	SRAM	>>	DRAM
power consumption (watts per bit):	RegFile	>>	SRAM	>>	DRAM

This is because, as we move from RegFiles to SRAMs to DRAMs, we use more and more highly specialized circuits for storage, and to multiplex data in and out of the storage arrays.

The idea of a cache is that we use fast (low latency), nearby memory as a temporary container for a subset of the memory locations of interest. When we want to read an address, if it is in the cache (a "hit"), we get the data quickly. If it is not in the cache (a "miss"), we must first invoke a process that brings that address from the backing memory into the cache, which may take many cycles, before we can service the original request. To do this we may also need to throw out or write back something else from the cache to the backing memory (which raises the question: which current item in the cache should we choose to replace with the new one?).

If, in a series of n accesses, most of them hit in the cache (we call this a high "hit rate"), then the overhead of processing a few misses is amortized over the many hits. Fortunately, most programs exhibit both spatial locality and temporal locality, which leads to high hit rates. Spatial locality means that if a program accesses an address A, it is likely to access nearby addresses (think of successive elements in an array, or fields in a struct, or sequences of instructions in instruction memory). Temporal locality means that much data is accessed more than once within a short time window (think of an index variable in a loop).

When we want to write an address, we are again faced with various choices. If that location is currently cached (a "write hit"), we should of course update it in the cache, but we could also immediately send a message to the backing memory to update its original copy; this policy is called *Write-through*. Alternatively, we could postpone updating the original location in backing memory until this address must be ejected from the cache (due to the need to use the space it occupies for some other cache line); this policy is called *Writeback*.

When writing, if the location is not currently cached, we could first bring the location into the cache and update it locally; this policy is called *Write-allocate*. Alternatively, we could just send the write to backing memory; this policy is called *Write-no-allocate* or *Write-around*.

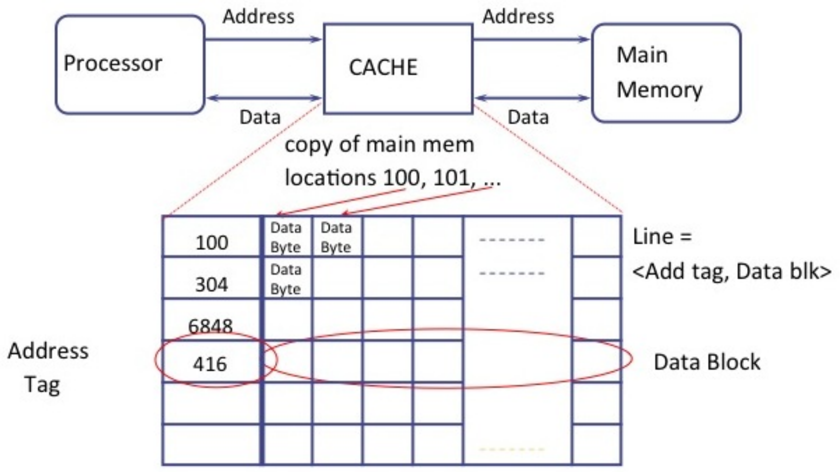


Figure 12.4: Sketch of the information in a cache

Fig. 12.4 illustrates the key information held in a cache. Each entry (each row shown) has an address tag, identifying to which (main memory) address A this entry corresponds. Along with this is a *cache line*, which is a vector of bytes corresponding to addresses A, $A+1$, $A+2$, ..., $A+k-1$ for some small fixed value of k. For example, a modern machine may have cache lines of 64 bytes. There are several reasons for holding data in cache-line-sized chunks. First, spatial and temporal locality suggest that when one word of the cache line is accessed, the other words are likely to be accessed soon. Second, the cost of fetching a line from main memory is so high that it's useful to amortize it over larger cache lines, and the buses and interconnects to memory are often more efficient in transporting larger chunks of data. Finally, The larger the cache line, the fewer things to search for in the cache (resulting in smaller tags, for example).

Conceptually, a cache is an *associative* or *content-addressable* memory, *i.e.,* to retrieve an entry with main memory address A we must search the cache for the entry with the address tag of A. A common software example of such a structure is a hash table. This poses an engineering challenge: it is not easy to build large *and* fast associative memories (we typically want a response in just 1 cycle if it is a cache hit). Solutions to this problem are the subject of this chapter.

Cache misses are usually classified into three types:

- *Compulsory misses*: The first time a cache line is accessed, it must be a miss, and it must be brought from memory into the cache. On large, long-running programs (billions of instructions), the penalty due to compulsory misses becomes insignificant.
- *Capacity misses*: This is a miss on a cache line that used to be in the cache but was ejected to make room for some other line, and so had to be retrieved again from backing memory.
- *Conflict misses*: Because large, fast, purely associative memories are difficult to build, most cache organizations have some degree of direct addressing using some of the bits of the candidate address. In this case, two addresses A_1 and A_2, where those bits are the same, may map into the same "bucket" in the cache. If A_1 is in the cache and we then want to read A_2, we may have to eject A_1, which may then be required again later; such a miss is called a conflict miss.

12.2 Cache organizations

Large, fast, purely associative memories are difficult to build. Thus, most caches compromise by incorporating some degree of direct addressing, using some of the bits of the target address as a direct address of the cache memories. Direct addressing and associative lookup can be mixed to varying degrees, leading to the following terminology:

- *Direct-Mapped caches* only use direct addressing. Some subset of the bits of the original address (typically the lower-order bits) are used directly as an address into the cache memory. Different addresses where the subset of bits have the same value will *collide* in the cache; only one of them can be held in the cache.

- *N-way Set-Associative caches* first, like direct-mapped caches, use some of the address bits directly to access the cache memory. However, the addressed location is wide enough to hold N entries, which are searched associatively. For different addresses where the subset of bits have the same value, N of them can be held in the cache without a collision.

Fig. 12.5 illustrates the organization of a direct-mapped cache. We use k bits (called the Index field) of the original address directly to address the 2^k locations of the cache memory. In that entry, we check the Valid bit (V); if invalid, the entry is empty, and we have a miss. If full, we compare the t bits of the Tag field of the address with the t bits of the Tag field of the entry, to see if it is for the same address or for some other address that happens to map to the same bucket due to having the same Index bits. If it matches, then we have a hit, and we use the b bits of the Offset field of the original address to extract the relevant byte(s) of the cache line.

Fig. 12.6 illustrates how we could have swapped the choice of t and k bits for the Index and Tag from the original address. In fact, any partitioning of the top $t + k$ bits into t and k bits would work. But it is hard to say which choice will result in a higher hit rate, since it depends on how programs and data are laid out in the address space.

Fig. 12.7 illustrates a 2-way set-associative cache. From a lookup point of view, it is conceptually like two direct-mapped caches in parallel, and the target may be found on

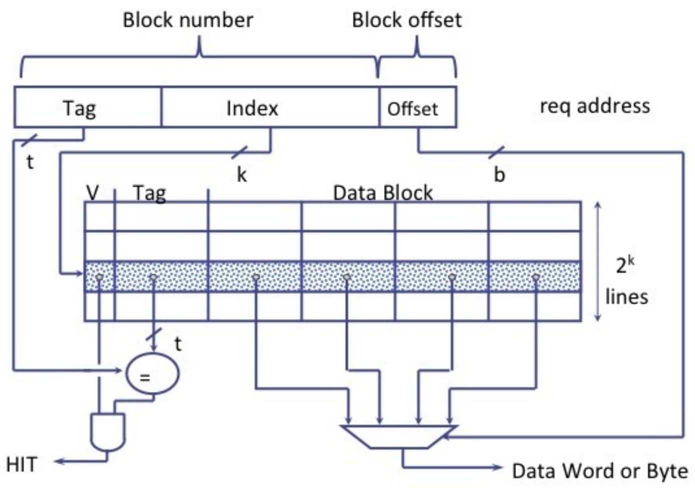


Figure 12.5: Direct-mapped cache organization

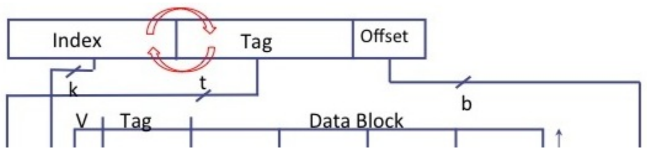


Figure 12.6: Choice of which bits to use in a direct mapped cache

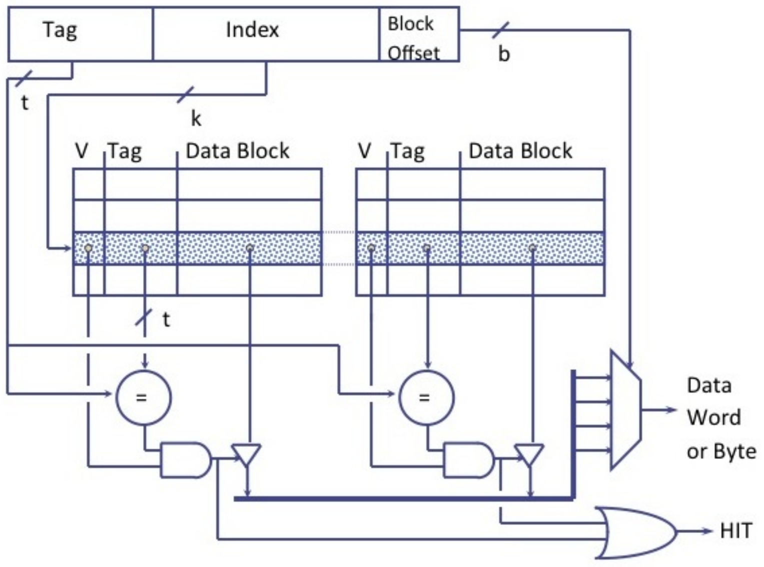


Figure 12.7: 2-way set-associative cache organization

either side. This can, of course, easily be generalized in principle to wider (4-way, 8-way, ...) associativity. But note that it costs more hardware: in particular, more comparators to test all k candidates in parallel.

The k entries at each location in the cache truly form a *set*, *i.e.,* we will be careful not to place duplicates in these entries, so that they represent distinct addresses, and do not waste cache resources. Thus, set-associativity is likely to improve the hit rate for the program, and thereby improve average memory access time which, in turn, should improve program performance. The average access time is given by the formula:

$$\text{average acces time} = \text{hit probability} \times \text{hit time} + \text{miss probability} \times \text{miss time}$$

12.2.1 Replacement policies

When a new cache line is to be brought in, usually it is necessary to eject another to make room for it. Which one should we eject? For direct-mapped caches, there is no choice—there is precisely one candidate entry. For k-way set-associative caches, we must choose one of the k candidates. There are many policies used in practice:

- Prefer lines that are “clean”, *i.e.,* lines that have been not been updated in the cache, and so are identical to the copy in backing store, and so can simply be thrown away. The complementary “dirty” lines must incur the overhead of writing them back to backing memory. Note that in Harvard architectures (where we do not dynamically modify code), cache lines in the instruction cache are always clean.
- A “fair” choice policy that does not always favor some members of the k-set over others. Such policies include LRU (Least Recently Used), MRU (Most Recently Used), Random, and Round-Robin. There is no “best” policy, since the effectiveness of each policy depends on how it interacts with the access patterns of a particular program. Only actual measurements on actual programs can determine what is a good policy choice (for that set of programs).

12.2.2 Blocking and Non-blocking caches

In some pipelined processors, when a Ld (load) instruction misses in the cache, it is possible that another Ld instruction following it closely in the pipeline also issues a request to the cache. What should the cache’s behavior be in this situation?

Blocking caches only service one memory request at a time, *i.e.,* they do not accept another request until they have completed the response to the previous one. In the case of a cache miss, this could be a delay of hundreds of cycles while it send a request and waits for a response from backing memory.

Non-blocking caches continue to accept requests while waiting to handle a miss on a previous request. These subsequent requests may actually hit in the cache, and the cache may be able to send that response immediately, out of order. Non-blocking caches typically have some bound on the number of requests that can simultaneously be in flight.

12.3 A Blocking Cache Design

In this section we will examine a design for a blocking cache with single-word cache lines, and with writeback and no-write-miss-allocate policies.

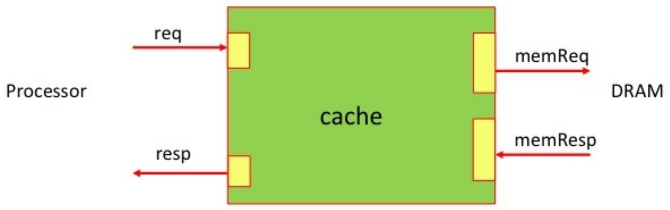


Figure 12.8: A cache interface

Fig. 12.8 shows the interface to a cache. In BSV:

Cache interface
```
interface Cache;
  method Action               req (MemReq r);
  method ActionValue #(Data) resp;

  method ActionValue #(MemReq) memReq;
  method Action                memResp (Line r);
endinterface
```

On the processor side of the cache, the processor sends a memory request (of type `MemReq`) to the cache using the `req` method. It retrieves a response (of type `Data`) from the cache using the `resp` method. On the memory side of the cache, the memory (or the bus) receives a memory request from the cache by using the `memReq` method (usually due to a cache miss), and returns a response to the cache using the `memResp` method. All methods are guarded so, for example, the `req` method will block until the cache yields a response via the `resp` method. The `memReq` will block until the cache is ready to issue a request to memory. The `req` to `resp` path essentially behaves like a 1-element FIFO: until a response is delivered ("dequeued"), the next request cannot be accepted ("enqueued").

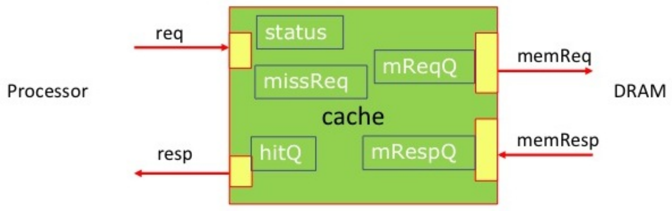


Figure 12.9: State elements in the cache

Fig. 12.9 shows the state elements of the cache, which are specified in the BSV code for the cache module, below.

Cache module
```
typedef enum { Ready, StartMiss, SendFillReq, WaitFillResp } CacheStatus
   deriving (Eq, Bits);
```

```

module mkCache (Cache);
   // Controls cache state machine
   Reg #(CacheStatus) status <- mkReg (Ready);

   // The cache memory
   RegFile #(CacheIndex, Line)              dataArray  <- mkRegFileFull;
   RegFile #(CacheIndex, Maybe #(CacheTag)) tagArray   <- mkRegFileFull;
   RegFile #(CacheIndex, Bool)              dirtyArray <- mkRegFileFull;

   FIFO #(Data)      hitQ    <- mkBypassFIFO; //  Allows 0-cycle hit responses
   Reg #(MemReq)     missReq <- mkRegU;

   FIFO #(MemReq) memReqQ  <- mkCFFIFO;
   FIFO #(Line)   memRespQ <- mkCFFIFO;

   function CacheIndex idxOf (Addr addr) = truncate (addr >> 2);
   function CacheTag   tagOf (Addr addr) = truncateLSB (addr);

   rule startMiss (status == StartMiss);
      let idx = idxOf (missReq.addr);
      let tag = tagArray.sub (idx);
      let dirty = dirtyArray.sub (idx);
      if (isValid (tag) && dirty) begin // writeback
         let addr = {validValue(tag), idx, 2'b0};
         let data = dataArray.sub (idx);
         memReqQ.enq (MemReq {op: St, addr: addr, data: data});
      end
      status <= SendFillReq;
   endrule

   rule sendFillReq (status == SendFillReq);
      memReqQ.enq (missReq);
      status <= WaitFillResp;
   endrule

   rule waitFillResp (status == WaitFillResp);
      let idx = idxOf (missReq.addr);
      let tag = tagOf (missReq.addr);
      let data = memRespQ.first;
      dataArray.upd (idx, data);
      tagArray.upd (idx, tagge Valid tag);
      dirtyArray.upd (idx, False);
      hitQ.enq (data);
      memRespQ.deq;
      status <= Ready;
   endrule

```

```
51    method Action req (MemReq r) if (status == Ready);
52       let idx = idxOf (r.addr);
53       let hit = False;
54       if (tagArray.sub (idx) matches tagged Valid .currTag
55           &&& currTag == tagOf (r.addr))
56          hit = True;
57       if (r.op == Ld) begin
58          if (hit)
59             hitQ.enq (dataArray.sub (idx));
60          else begin
61             missReq <= r;
62             status <= StartMiss;
63          end
64       end
65       else begin // It is a store request
66          if (hit) begin
67             dataArray.upd(idx, r.data);
68             dirtyArray.upd (idx, True);    // for a 1-word cache line
69          end
70          else
71             memReqQ.enq(r); // write-miss no allocate
72       end
73    endmethod
74
75    method ActionValue#(Data) resp;
76       hitQ.deq;
77       return hitQ.first;
78    endmethod
79
80    method ActionValue#(MemReq) memReq;
81       memReqQ.deq;
82       return memReqQ.first;
83    endmethod
84
85    method Action memResp (Line r);
86       memRespQ.enq(r);
87    endmethod
88 endmodule
```

The cache behavior is a state machine that normally rests in the Ready state. On a miss, it goes into the StartMiss state, sending a fill request (and possibly a writeback) to memory. It waits for the response in the WaitFillResp state. When the response arrives, it performs final actions, including sending a response to the processor, and returns to the Ready state. The type for this state is defined on lines 1 and 2.

The cache memory itself is implemented on lines 9-11 using `mkRegFileFull` for pedagogical convenience; more realistically, it would be implemented with an SRAM module. The `hitQ` on line 13 is a `BypassFIFO` to enable a 0-cycle (combinational) response on hits.

More realistic caches typically do not give a 0-cycle response, in which case this can become an ordinary FIFO or a `PipelineFIFO`.

The memory request and response FIFOs on lines 16-17 are conflict-free FIFOs because they totally isolate the cache from the external memory system, and they are in any case not performance-critical. The two functions on lines 19-20 are just convenience functions to extract the appropriate bits from index and tag bits of an address.

The `req` method only accepts requests in the Ready state. It tests for a hit on lines 54-55 by indexing the tag array, checking that it is valid, and checking that the entry's tag matches the tag of the address. Line 59 handles a hit for a Load request; the response is entered in `hitQ`. Lines 61-62 start the miss-handling state machine. Line 67-68 handle Store requests that hit. The data is stored in the cache entry and the entry is marked dirty (remember, we are implementing a writeback policy, and so this write is not communicated to memory until the line is ejected). This example code assumes one `Data` word per cache line; for longer cache lines, we'd have to selectively update only one word in the line. In the write-miss case, we merely send the request on to memory, *i.e.,* we do not allocate any cache line for it.

The remaining methods are just interfaces into some of the internal FIFOs. The processor receives its response from `hitQ`.

The `startMiss` rule on line 22 fires when the `req` method encounters a miss and sets `state` to `StartMiss`. It checks if the current cache entry is dirty and, if so, writes it back to memory. Finally, it enters the `SendFillReq` state, which is handled by the `sendFillReq` rule which sends the request for the missing line to memory. It waits for the response in the `WaitFillResp` state. When the response arrives, the `waitFillResp` rule stores the memory response in the cache entry and resets its dirty bit, enters the response into `hitQ`, and returns to the `Ready` state.

Note that in case of a Load-miss, if the cache line is not dirty, then the `startMiss` rule does nothing (line 29 is not executed). In general, a few idle cycles like this don't seriouly affect the miss penalty (if it is important, it is possible to code it to avoid idle cycles like this).

12.4 Integrating caches into the processor pipeline

Integrating the Instruction and Data Caches into the processor pipeline is quite straightforward. The one issue that might bear thinking about is that the caches have a variable latency. Each cache behaves like a FIFO for Loads: we "enqueueue" a Load request, and we later "dequeue" its response. In the case of a hit, the response may come back immediately; whereas in the case of miss, the response may come back after many (perhaps 100s) of cycles. But this is not a problem, given our methodology from the beginning of structuring our pipelines as elastic pipelines that work correctly no matter what the delay of any particular stage.

Fig. 12.10 shows our 4-stage pipeline. The orange boxes are FIFOs that carry the rest of the information associated with an instruction (instruction, pc, ppc, epoch, poison, ...) while a request goes through the corresponding cache "FIFO". In other words, we are conceptually forking the information; one part goes through our in-pipeline FIFO and the

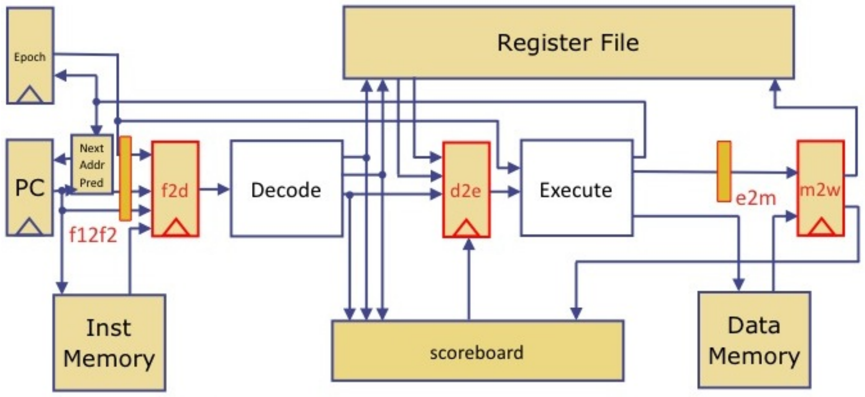


Figure 12.10: Integrating caches into the pipeline

other part goes through the cache. These pieces rejoin on the other side of the FIFOs. Because of the elastic, flow-controlled nature of all the components, all of this just works.

The one nuance we might consider is that, if the caches can return responses in 0 cycles (combinationally, on a hit), then the accompanying FIFOs in the pipeline should be BypassFIFOs, to match that latency.

12.5 A Non-blocking cache for the Instruction Memory (Read-Only)

In this section we show a non-blocking cache for the Instruction Memory. Remember that in Harvard architectures, Instruction Memory is never modified, so there are no Store requests to this cache, nor are there ever any dirty lines. Non-blocking caches for Data Memory (Read-Write) are significantly more complicated, and we discuss them briefly in the Conclusion section.

Fig. 12.11 shows the structure of the non-blocking cache. The key new kind of element is the “Completion Buffer” (CB), on the top right of the diagram. A CB is like a FIFO. The `reserve` method is like an “early” enqueue operation, in that we reserve an ordered place in the FIFO, with a promise to deliver the data later using the `complete` method. When we reserve the space, the CB gives us a “token” with our reservation, which is essentially a placeholder in the FIFO. When we later complete the enqueue operation, we provide the data and the token, so that the CB knows exactly where to place it in the FIFO. The `drain` method is just like dequeing the FIFO: it drains it in order, and will block until the actual data has arrived for the first reserved space in the FIFO. The Completion Buffer is a standard package in the BSV library.

Below, we show the BSV code for the non-blocking read-only cache. The completion buffer is instantiated in line 6. The parameter 16 specifies the capacity of the completion buffer, *i.e.,* which allows the non-blocking cache to have up to 16 requests in flight.

The `req` method first reserves a space in the cb fifo. Then, if it is a hit, it completes the operation immediately in the cb fifo. If it is a miss, it sends the request on to memory, and enqueues the token and the request in the `fillQ`, in lines 35-36. Note that the `req` method is immediately available now to service the next request, which may be a hit or a miss.

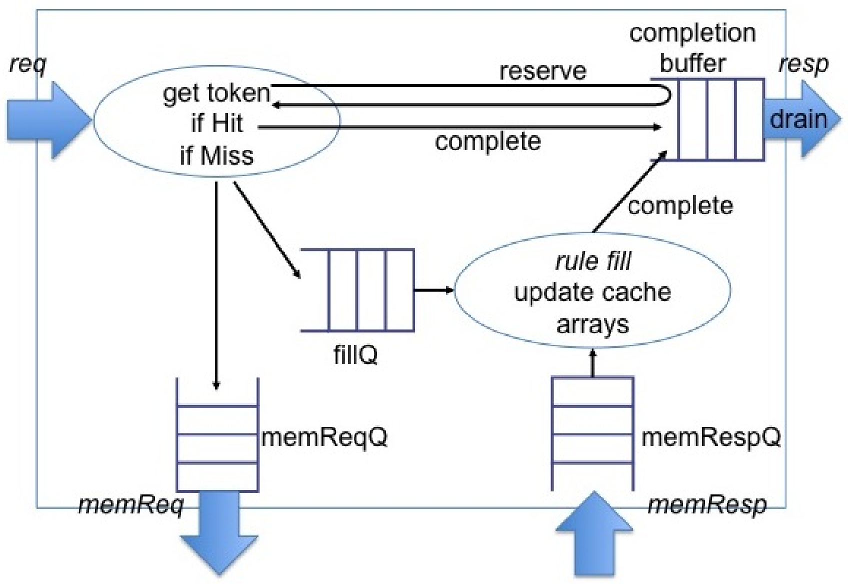


Figure 12.11: Non-blocking cache for Instruction Memory

Independently, when a fill response comes back from memory, the `fill` rule collects it and the information in the `fillQ`, updates the cache information and then invokes the `complete` method on the completion buffer. In the meanwhile, the `req` method could have accepted several more requests, some of which may be hits that are already in the completion buffer.

The `resp` method returns the responses in the proper order by using the `drain` method of the completion buffer.

Non-blocking Read-Only Cache module

```
module mkNBCache (Cache);
   // The cache memory
   RegFile #(CacheIndex, Line)               dataArray  <- mkRegFileFull;
   RegFile #(CacheIndex, Maybe #(CacheTag)) tagArray   <- mkRegFileFull;

   CompletionBuffer #(16, Data) cb <- mkCompletionBuffer;
   FIFO #(Tuple2 #(Token, MemReq)) fillQ <- mkFIFO;

   FIFO #(MemReq) memReqQ  <- mkCFFIFO;
   FIFO #(Line)   memRespQ <- mkCFFIFO;

   function CacheIndex idxOf (Addr addr) = truncate (addr >> 2);
   function CacheTag   tagOf (Addr addr) = truncateLSB (addr);

   rule fill;
      match  .token, .req  = fillQ.first; fillQ.deq;
      data = memRespQ.first; memRespQ.deq;

      let idx = idxOf (req.addr);
      let tag = tagOf (req.addr);
      dataArray.upd (idx, data);
      tagArray.upd (idx, tagged Valid tag);

      cb.complete.put (tuple2 (token, dataArray.sub (idx)))
   endrule

   method Action req (MemReq r);
      let token <- cb.reserve.get;
      if (tagArray.sub (idxOf (r.addr)) matches tagged Valid .currTag
          &&& currTag == tagOf (r.addr))
         // hit
         cb.complete.put (token, dataArray.sub (idx));
      else begin
         // miss
         fillQ.enq (tuple2 (token, r));
         memReqQ.enq (r);    // send request to memory
      end
   endmethod

   method ActionValue#(Data) resp;
      let d <- cb.drain.get
      return d;
   endmethod

   method ActionValue#(MemReq) memReq;
      memReqQ.deq;
      return memReqQ.first;
   endmethod

   method Action memResp (Line r);
      memRespQ.enq (r);
   endmethod
endmodule
```

How big should the `fillQ` be? Suppose it has size 2. Then, after the `req` method has handled 2 misses by sending 2 requests to memory and enqueuing 2 items into `fillQ`, then `fillQ` is full until the first response from memory returns and rule `fill` dequeues an item from `fillQ`. Method `req` will then be stuck on the next miss, since it will be blocked from enqueueing into `fillQ`. Thus, the capacity of `fillQ` is a measure of how many requests to memory can be outstanding, *i.e.,* still in flight without a response yet. This choice is a design decision.

12.5.1 Completion Buffers

Here we show the code for a Completion Buffer, which is quite easy to implement using CRegs. First, here are some type definitions for completion buffer tokens (FIFO placeholders) and the interface. The interface is parameterized by n, the desired capacity, and t, the type of data stored in the buffer.

Completion Buffer token and interface types

```
typedef struct    UInt #(TLog #(n))  ix;    CBToken2 #(numeric type n)
deriving (Bits);

interface CompletionBuffer2 #(numeric type n, type t);
   interface Get #(CBToken2 #(n))               reserve;
   interface Put #(Tuple2 #(CBToken2 #(n), t))  complete;
   interface Get #(t)                           drain;
endinterface
```

Completion Buffer module

```
module mkCompletionBuffer2 (CompletionBuffer2 #(n, t))
   provisos (Bits #(t, tsz),
             Log #(n, logn));

   // The following FIFOs are just used to register inputs and outputs
   // before they fan-out to/fan-in from the vr_data array
   FIFOF #(Tuple2 #(CBToken2 #(n), t))  f_inputs  <- mkFIFOF;
   FIFOF #(t)                           f_outputs <- mkFIFOF;

   // This is the reorder buffer
   Vector #(n, Reg #(Maybe #(t))) vr_data <- replicateM (
                                                 mkReg (tagged Invalid));

   // Metadata for the reorder buffer (head, next, full)
   Reg #(Tuple3 #(UInt #(logn),
                  UInt #(logn),
                  Bool)         cr_head_next_full [3]
                                   <- mkCReg (3, tuple3 (0,0, False));

   match  .head0, .next0, .full0  = cr_head_next_full [0];
   match  .head1, .next1, .full1  = cr_head_next_full [1];

   function UInt #(logn) modulo_incr (UInt #(logn) j);
      return ( (j == fromInteger (valueOf (n) - 1)) ? 0 : j+1);
   endfunction

   rule rl_move_inputs;
      let t2 = f_inputs.first; f_inputs.deq;
      match  .tok, .x  = t2;
      vr_data [tok.ix] <= tagged Valid x;
   endrule

   rule rl_move_outputs (vr_data [head0] matches tagged Valid .v
                         &&& ((head0 != next0) || full0));
      vr_data [head0] <= tagged Invalid;
      cr_head_next_full [0] <= tuple3 (modulo_incr (head0), next0, False);
      f_outputs.enq (v);
   endrule

   // ----------------------------------------------------------------
   // Interface

   interface Get reserve;
      method ActionValue #(CBToken2 #(n)) get () if (! full1);
         let next1_prime = modulo_incr (next1);
         cr_head_next_full [1] <= tuple3 (head1,
                                          next1_prime,
                                          (head1==next1_prime));
         return CBToken2  ix: next1 ;
      endmethod
   endinterface

   interface complete = toPut (f_inputs);
   interface drain    = toGet (f_outputs);
endmodule
```

12.6 Conclusion

In the previous section, even though the non-blocking cache could accept and service multiple requests, it returned responses in standard FIFO order. More sophisticated processor designs will expose the out-of-order responses into the processor, *i.e.,* the cache would return responses immediately, and the processor may go ahead to execute those instructions if it were safe to do so. In this case, the processor itself will associate a token with each request, and the cache would return this token with the corresponding response; this allows the processor to know which request each response corresponds to.

Non-blocking caches for the Data Memory are more complex, because we can then encounter RAW (Read-After-Write) hazards corresponding to memory locations, analogous to the RAW hazards we saw in Chapter 9 that concerned register file locations. For example, suppose the cache gets a read request Q_1 which misses; it initiates the request to memory, and is pending the response. Suppose now it gets a write request Q_2 for the same cache line. Clearly it must wait for the pending memory read response; it must respond to the processor for Q_1 based on the data returned from memory; and only then can it process Q_2 by updating the cache line. If we allow several such interactions to be in progress across different cache lines, the design becomes that much more complex.

In this chapter we have barely scratched the surface on the subject of caches. High-performance memory systems, which revolve around clever cache design, is one of the major subjects in Computer Architecture, since it is such a major determinant of program and system performance.

Chapter 13

Virtual Memory

13.1 Introduction

Almost all modern general-purpose processors implement a virtual memory system. Virtual memory systems, sketched in Fig. 13.1, simultaneously addresses many requirements.

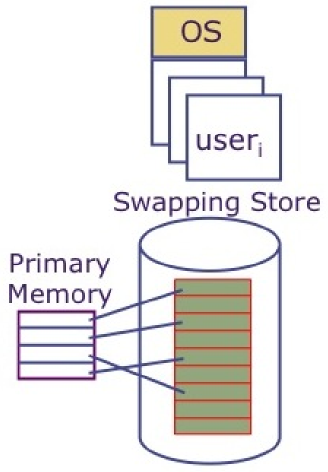


Figure 13.1: Virtual Memory by Paging between physical memory and secondary store

Large address spaces: many applications need large address spaces for the volume of data to which they need random access. Further, modern computer systems often multiplex amongst several such applications. Virtual memory systems permit these larger "memories" actually to reside on large, cheap secondary stores (primarily magnetic disk drives), using a much smaller, but more expensive, DRAM memory as a cache for the disk. For historical reasons, this DRAM memory is also called *core memory*[1] or *physical memory*.

Just as an SRAM cache temporarily holds cache lines from DRAM, in virtual memory systems DRAM temporarily holds *pages* from secondary store. Whereas cache line sizes are typically a few tens of bytes (e.g., 64 bytes), page sizes typically range from 512 Kilobytes

[1]Since the days when memories were implemented using magnetic cores (principally the 1960s and 1970s).

(small) to 4 or 8 kilobytes (more typical). The disk is also known as a *swapping* store, since pages are swapped between physical memory and disk.

Protection and Privacy: Even in a single user system, the user's application should not be allowed, inadvertently or maliciously, to damage the operating systems code and data structures that it uses to manage all the devices in the system. In a multi-user system, one user's application program should not be allowed to read or damage another users's application code or data.

Relocatability, dynamic libraries and shared dynamic libraries: We should be able to generate machine code for a program independently of where it is loaded in physical memory; specifically, the PC target information in the branch instructions should not have to be changed for each loading. For several years we have also started dynamically loading program libraries on demand during program execution. And, more recently, dynamic libraries have been shared across multiple applications running at the same time, and they may not be loadable at the same address in different applications.

Paging systems for virtual memory enable a solution for these and many similar requirements.

13.2 Different kinds of addresses

Fig. 13.2 shows various kinds of addresses in a system with virtual memory.

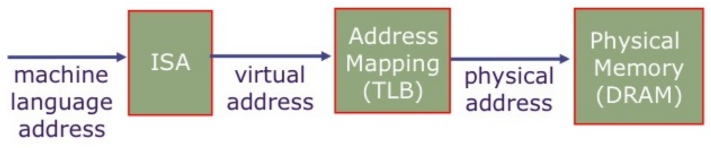


Figure 13.2: Various kinds of addresses in a Virtual Memory system

A *machine language address* is the address specified in machine code.

A *virtual address* (VA), sometimes also called an *effective address*, is a mapping from the machine language address specified by the ISA. This might involve relocation from a base address, or adding process identifier bits, segment bits, etc.

A *physical address* (PA) is an address in the physical DRAM memory. The virtual address is translated into a physical address through a data structure called a *Translation Lookaside Buffer* (TLB). The actual translation is always done in hardware, the operating system is responsible for loading the TLB.

13.3 Paged Memory Systems

Fig. 13.3 shows how VAs are translated to PAs using a *page table*. At the top of the figure we see how a virtual address can be interpreted as a *virtual page number* (VPN) and an *offset* within the identified page. For example, if our page sizes are 4 KB, then the lower 12 bits specify the offset and the remaining bits specify the page number. Physical memory is also divided into page-sized chunks. The VPN is used as an index into the *page table*,

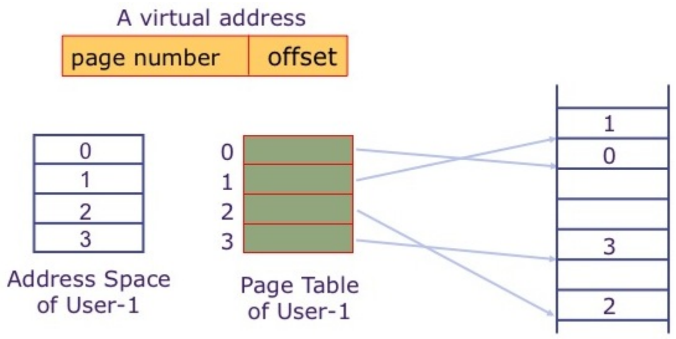


Figure 13.3: Mapping VAs to PAs using a page table

where each entry is the physical address of a page in physical memory, *i.e.,* a *Physical Page Number* (PPN). The specified location is then accessed at the given offset within that page of physical memory. As suggested by the criss-crossing lines in the diagram, this level of indirection through the page table also allows us to store virtual pages non-contiguously in physical memory.

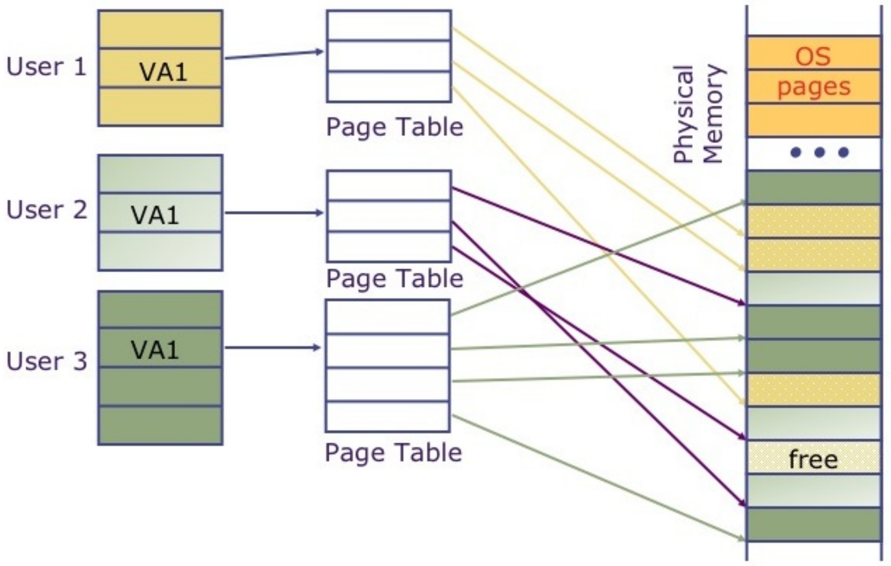


Figure 13.4: Multiple address spaces using multiple page tables

The page table indirection also enables us to share the physical memories between the Operating System and multiple users. Each such process has its own page table that maps its addresses to its pages in physical memory. In fact the mapping does not have to be disjoint: if many users are running the same application (e.g., the same web browser), we could have a single copy of the code for the web browser in physical memory, and in each page table the part of it that holds the web browser program code could point to the same, shared physical pages.

13.4 Page Tables

If we have 32-bit virtual addresses, we would need 4 Gigabytes of disk space to hold the entire address space. Disks of this size are now feasible (they were not, a decade and more ago), but of course many modern ISAs use 64-bit virtual addresses, for which it is simply infeasible to actually hold the entire virtual address space on disk. But in fact programs don't actually use that much data, so in fact page tables are usually *sparse*, *i.e.,* many entries are simply "empty".

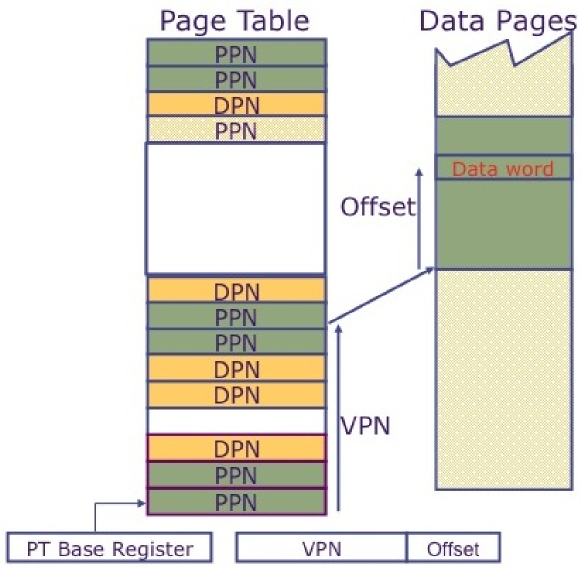


Figure 13.5: A simple linear page table

Fig. 13.5 shows a simple linear page table. Each entry in the table, called a *Page Table Entry* (PTE) typically contains at least the following information:

- A bit that indicates whether it is empty or not (valid bit).
- If valid and currently mapped to physical memory, the *Physical Page Number* (PPN) identifying the physical memory page where it is mapped.
- A *Disk Page Number* identifying the "home" page on disk where the page resides when it is not mapped into physical memory.
- Additional status and protection bits. For example, to implement the idea of a Harvard architecture, pages that contain program code may be marked "read-only"; if there is an attempt to write such page, then an exception is raised during the page table lookup.

The VPN of the virtual address is used as an index into the page table to retrieve a PTE, which is then used to access the desired word at the offset in the virtual address. Since page tables are set up by the operating system, it makes sense for page tables themselves to reside in main memory, instead of in a separate, special memory. The *PT Base Register* shown in the diagram is used to allow the OS to place the page table at a convenient address.

Page tables can be large. For example, with a 32-bit address space and 4 KB page size (= 12-bit offset), the page number is 20 bits. If each PTE takes 4 bytes, the page table will be 2^{22} = 4 MB. If we have multiple users and multiple processors running on the machine,

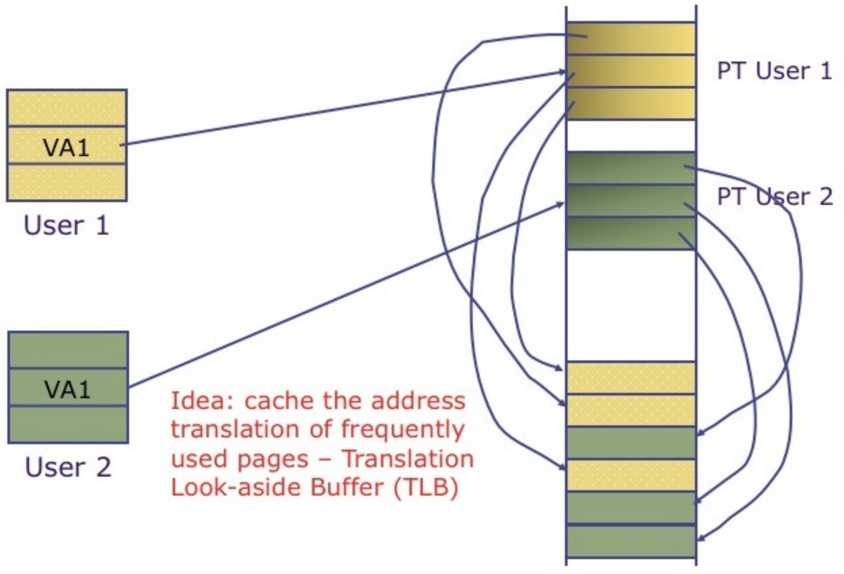

VA1
User 1
VA1
User 2
PT User 1
PT User 2
Idea: cache the address translation of frequently used pages – Translation Look-aside Buffer (TLB)

Figure 13.6: Page table for multiple address spaces in physical memory

their combined page tables themselves may run into multiple Gigabytes. The only place such large arrays will fit is in physical memory itself, as illustrated in Fig.13.6.

Of course, if we used larger pages, the page number would use fewer bits, and this would reduce the size of the page table. But larger pages have two problems. First, it may result in so-called *internal fragmentation*, where large parts of a page are unused (for example if we place each subroutine in a shared library in its own page). Second, it increases the cost of swapping a page between disk and physical memory (called the *page fault penalty*). The problem gets worse in ISAs with 64-bit addresses. Even a 1 MB page size results in 44-bit page numbers and 20-bit offsets. With 4-byte PTEs, the page table size is 2^{46} bytes (70 Petabytes)! It is clearly infeasible actually to lay out such a large page table.

Fortunately, a 64-bit address space is very sparsely used, *i.e.,* large tracts of the page table would contain “empty” PTEs. A hierarchical (tree-like) page table is a more efficient data structure for sparse page tables, and this is illustrated in Fig. 13.7. Entire subtrees representing large, contiguous, unused regions of the address space are simply “pruned” from the tree.

13.5 Address translation and protection using TLBs

Fig. 13.8 shows the activities involved in translating a virtual address into a physical address. The inputs are: a virtual address, whether it is a read or a write access, and whether the request was issued by a program in kernel mode or user mode (more generally, some indication of the “capability” of the requestor to access this particular page). The VPN is looked up in the page table, as we have discussed. The read/write and kernel/user attributes are checked against the protection bits in the PTE, and if there is a violation, an exception is raised in the processor pipeline instead of performing the memory operation.

Note that these activities are performed on *every* memory access! It has to be efficient

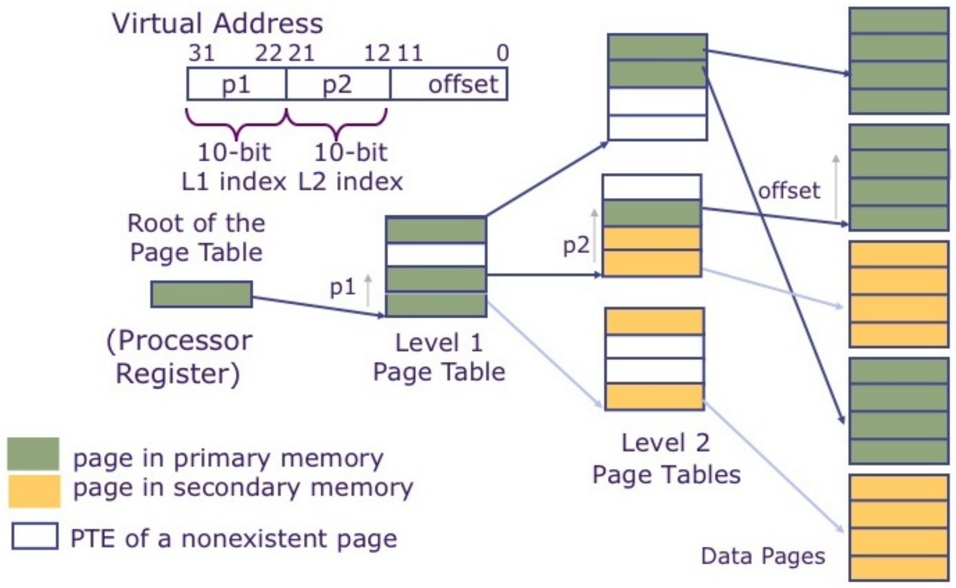


Figure 13.7: Hierarchical page tables for large, sparse address spaces

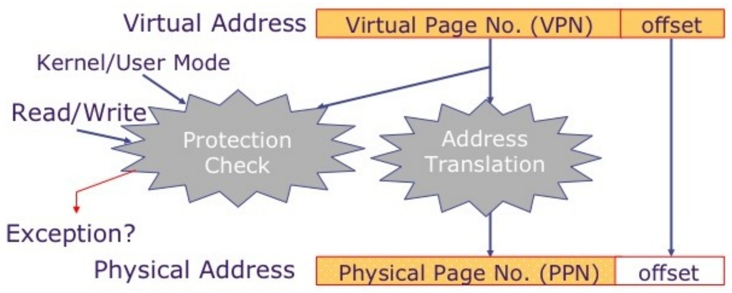


Figure 13.8: Address translation and protection

if we are to maintain program performance! For example, it is infeasible to emulate this in software; it must be done in hardware. Even in hardware, with a linear page table, each logical memory reference becomse two physical memory references, one to access the PTE and then one to access the target word. The solution, once again, is to cache PTEs in small, fast, processor-local memory. These caches are called *Translation Lookaside Buffers* (TLBs).

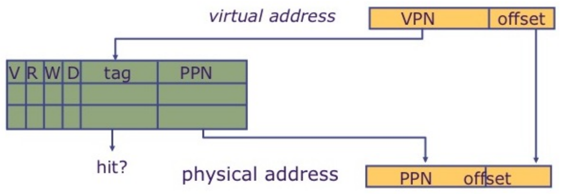


Figure 13.9: A Translation Lookaside Buffer

Fig. 13.9 shows a TLB, which is typically a very small local memory (at most a few tens of PTEs, typically). It is associatively searched for the current VPN. If successful (a hit), it immediately yields the corresponding PPN (and the protection checks can be done as well). Otherwise (it is a miss), we need to suspend operations, load the corresponding PTE, and then complete the original translation. This is called a *Page Table Walk*, since modern page tables are hierarchical and finding the PTE involves traversing the tree, starting from the root.

TLBs typically have from 32-128 entries, and are fully associative. Sometimes larger TLBs (256-512 entries) are 4- or 8-way set-associative. The *TLB Reach* is a measure of the largest chunk of virtual address space that is accessible from entries in the TLB. For example, with 64 TLB entries and 4 KB pages, the program can access $64 \times 4 = 256$ KB of the address space (may not be contiguous).

On a miss, which entry to eject? A random or LRU (least-recently used) policy is common. Address mappings are typically fixed for each address space, so PTEs are not updated, so there is no worry about dirty entries having to be written back.

13.6 Variable-sized pages

We mentioned earlier that a uniformly large page size, though it can shrink the page table and increase the TLB hit rate, can lead to internal fragmentation and large page fault penalties. However, there are plenty of programs that can exploit large, densely occupied, contiguous regions of memory. Thus, one strategy is to support multiple page sizes. This is illustrated in Fig. 13.10.

In the diagram, a 32-bit Virtual Page Address is interpreted either as having a 20-bit VPN for a 4 KB page as before, or a 10-bit VPN for a 4 MB page (variously called “huge pages”, “large pages” or “superpages”). In the translation process, we use 10 bits (p1) to access the root of the page table tree, the Level 1 page table. Information in the PTE tells us whether it directly points at a large page, or whether it points to a Level 2 page table where the remaining 10 bits (p2) will be used to locate a normal page.

The TLB has to change accordingly, and this is illustrated in Fig. 13.11. Each entry in the TLB has an extra bit indicating whether it is for a normal page or for a large page. If

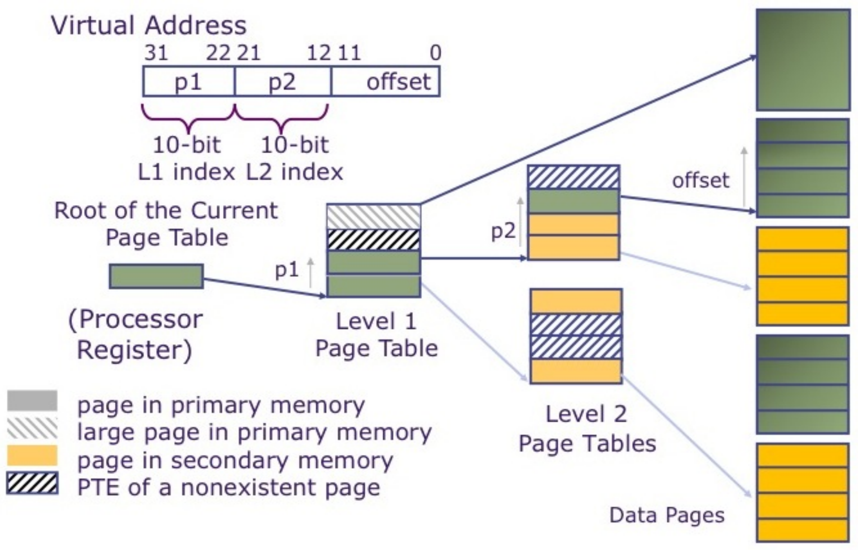


Figure 13.10: Variable-sized pages

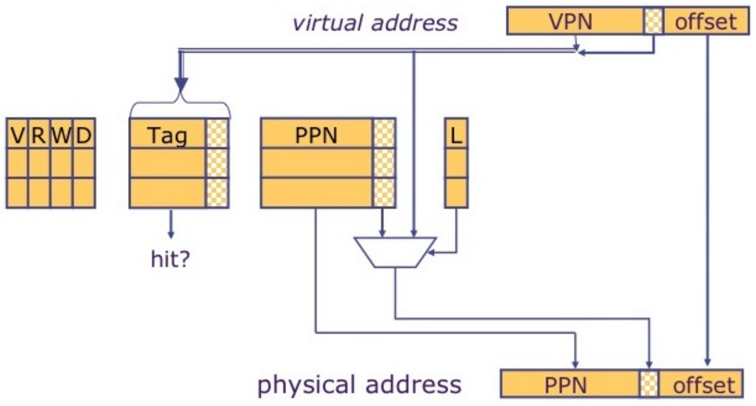


Figure 13.11: TLB for variable-sized pages

for a large page, then only the first 10 bits are matched; the TLB yields a short PPN, and the remaining address 10 bits are passed through to be concatenated into the offset. If for a normal page, then all 20 bits are matched, and the TLB yields the longer PPN for the normal page.[2]

13.7 Handling TLB misses

The page table walk may be done in hardware or in software. For example, in the DEC Alpha AXP ISA (in the 1990s) and in the MIPS ISA, a TLB miss exception traps into a software handler which fixes up the TLB before retrying the provoking instruction. The handler executes in a privileged, "untranslated" addressing mode to perform this function (since we don't want the handler itself to encounter page faults).

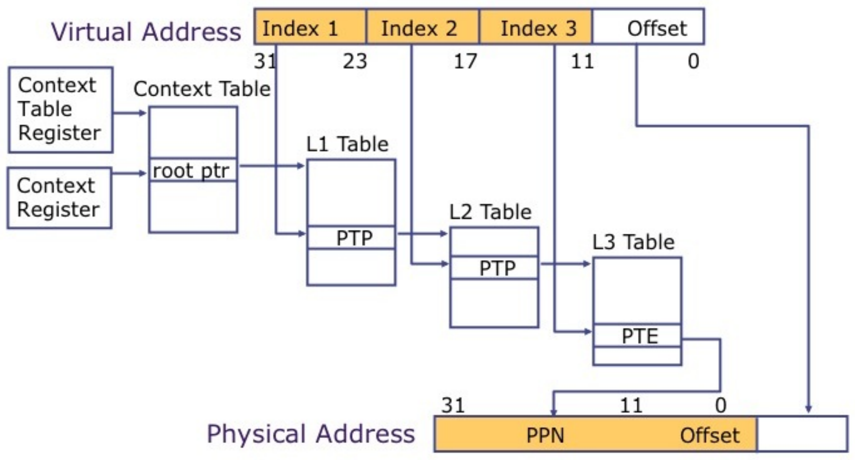


Figure 13.12: Hierarchical Page Table Walk in SPARC v8

In the SPARC v8, x86, PowerPC and ARM ISAs, the TLB walk is performed in a hardware unit called the Memory Management Unit (MMU). The hardware for a tree walk is a relatively simple state machine. If a missing data or page table page is encountered by the hardware, then the MMU abandons the attempt and raises a page-fault exception for the original memory access instruction.

Fig. 13.12 shows the hierarchical page table walk in the MMU in the SPARC v8 architecture. It is essentially a sequential descent into the page table tree using various bits of the virtual address to index into a table at each leve to retrieve the address of a sub-tree table.

Fig. 13.13 summarizes the full address translation activities. We start with a TLB lookup. If it's a hit, we do the protection check and, if permitted, we have the physical address and can perform the memory access. If there is a protection violation, we raise a "segfault" (segment fault) exception. If the TLB lookup was a miss, we do the Page Table Walk (in hardware or in software). If the page is in physical memory, we update the TLB and complete the memory operation (protection check, etc.). If the page is not in physical

[2]Note: large and short pages will be aligned to 22 bit and 12 bit addresses, respectively, so their PPNs will have correspondingly different sizes.

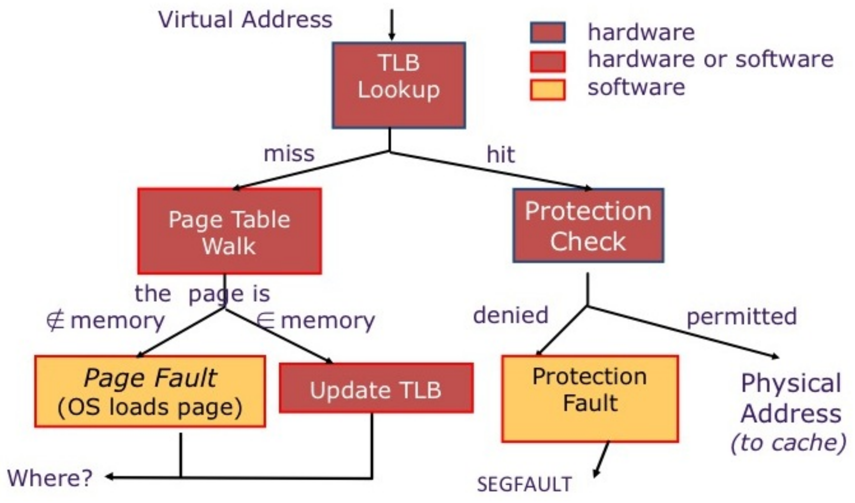


Figure 13.13: Putting it all together

memory, we raise a Page Fault exception, which traps to the operating system to swap the page in from disk, after which the original memory access can be retried.

13.8 Handling Page Faults

When we encounter a page fault, *i.e.,* a memory request for a word in a virtual page that currently only resides on disk and is not currently swapped into a physical page, the handler must perform a swap operation. The missing page must be located (or allocated) on disk. It must be copied into physical memory; the PTE for this page in the page table must be updated; and, the TLB must be loaded with this entry.

We call it a "swap" because, as is generally true in cacheing, this may require another page that is currently in physical memory to be written back to disk in order to make room for this one. We have many of the same considerations as in cacheing:

- Only dirty pages need to be written back to disk, *i.e.,* pages whose copies in physical memory have been updated (written) since they were brought in from disk.
- The policy for selecting the "victim" page that will be overwritten (and, if dirty, first ejected back to disk). Again, possible policies include: favoring clean pages over dirty pages, random, LRU, MRU, round-robin, and so on.

Page fault handling is always done in software. Moving pages between physical memory and disk can take milliseconds, which is extremely slow compared to modern processor speeds; and well-behaved programs don't page fault too often, and so doing it in software has acceptable overheads and performance. The actual movement of page contents between disk and physical memory is usually performed by a separate hardware component called a Direct Memory Access engine (DMA): the processor merely initializes the DMA with the information about the size of the transfer and its locations in memory and disk, and the DMA engine then independently performs the copy, usually generating an interrupt to the processor when it has completed. While the DMA is doing its work, the processor switches to some other process to do some useful work in the interval, instead of waiting for this operation.

13.9 Recursive Page Faults

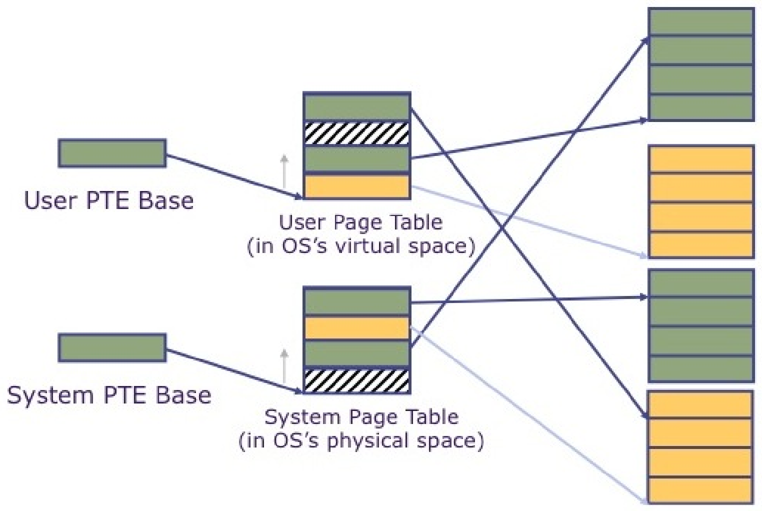


Figure 13.14: Recursive problem of TLB misses and pages faults during PT walk

During a page table walk, the MMU (or a software handler) is itself accessing memory (for the page tables). If the page tables are themselves in the operating system's virtual address space, these accesses may, in turn, miss in the TLB, which itself needs a page table walk to refill. During this walk, we may again encounter a page fault. This is a kind of recursive page fault problem. The situation (and the solution) is illustrated in Fig. 13.14. As usual in recursive situation, we need a "base case" to terminate the recursion. We do this by placing the the page table for the system in physical address space which requires no translation. The TLB miss handler, and the page fault handler, for this page table run in a special privileged mode that works directly on physical addresses. Of course, the region of physical memory used for these "pinned" data structures and codes is a special reserved region of physical memory that is not part of the pool of page frames where we swap virtual pages. The partitioning of available physical memory (which varies across computer systems) into these regions is typically done by the operating system at boot time.

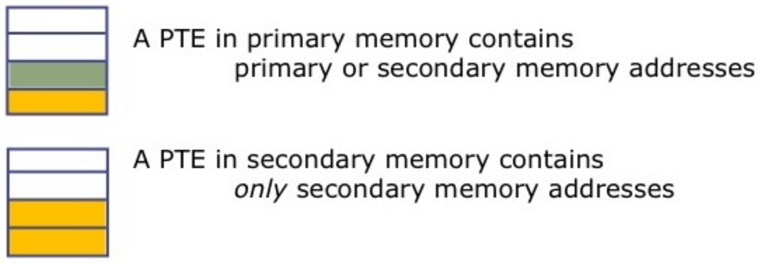

A PTE in primary memory contains
primary or secondary memory addresses
A PTE in secondary memory contains
only secondary memory addresses

Figure 13.15: Pages of a page table

One issue about which page fault handlers must take care is illustrated in Fig. 13.15. Consider a page P_{PT} of a page table that itself resides in virtual memory. When it is in physical memory, some of its PTEs will contain physical addresses of the pages P_1, $_2$, ... which it is responsible—those final address-space pages that have also currently been swapped in. We do not want to swap out P_{PT} while P_1, P_2, ... are still swapped in—that would mean we would encounter a page fault for accesses to those pages even though they are resident in physical memory. Thus, we swap out P_PT only after P_1, P_2, ... have themselves been swapped out. A corollary to this is that when P_PT resides on disk, all its PTEs only contain disk addresses, never physical memory addresses.

13.10 Integating Virtual Memory mechanisms ino the processor pipeline

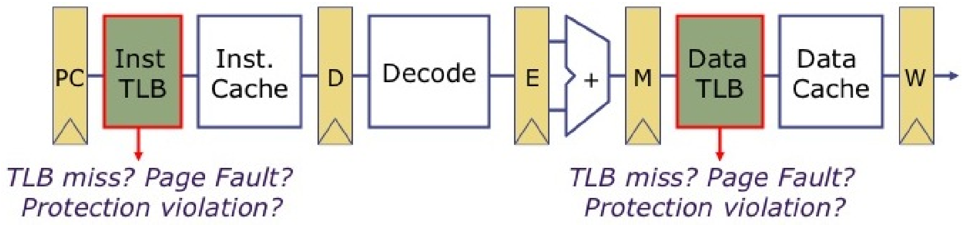

PC
Inst TLB
Inst. Cache
D
Decode
E
+
M
Data TLB
Data Cache
W
TLB miss? Page Fault? Protection violation?
TLB miss? Page Fault? Protection violation?

Figure 13.16: Virtual Memory operations in the processor pipeline

Fig. 13.16 shows virtual memory operations in the processor pipeline. In two places—when fetching an instruction from memory, and when accessing memory for a Ld or St instruction— we may encounter TLB misses, protection violations and page faults. Each of these raises an exception and is handled using the mechanisms we studied in Ch. 11 on exceptions. These exceptions are *restartable* exceptions, i.e, the operation (instruction fetch, Ld/St instruction execution) is retried after the exception has been handled (unlike, say, a divide-by-zero exception, where we do not retry the instruction).

In the TLB boxes in the diagram, how do we deal with the additional latency needed for the TLB lookup? We can slow down the clock, giving more time for the TLB lookup operation. We can pipeline the TLB and cache access (if done in that order, the caches work on physical addresses). We can do the TLB access and cache access in parallel (in which case the cache operates on virtual addresses).

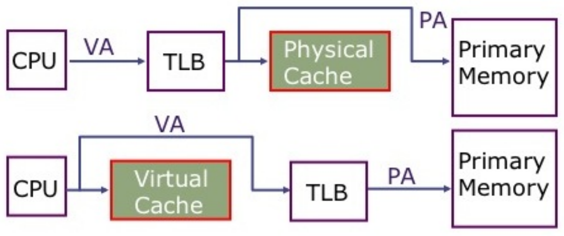


Figure 13.17: Caches based on Virtual or Physical Addresses

Fig. 13.17 shows two different organizations of TLB lookup and cache lookup. In the first case the cache works on the physical addresses that come out of TLB translation. In the second case the cache works immediately on the virtual address from the processor pipeline. The advantage of the latter is speed: the cache lookup is initiated earlier, and in the case of a hit, the processor pipeline can proceed immediately. But it has some disadvantages as well.

First, since all processes (contexts) have the same virtual address space, we must flush the cache (empty it out completely) whenever there is a context switch. Otherwise, if address 42 from process A was in the cache, then we would accidentally have a false hit when process B tries to access its address 42, which in general has nothing to do with A's address 42. An alternative is to attach some process identifier bits to the cache tag of address, so that tags will never match across contexts.

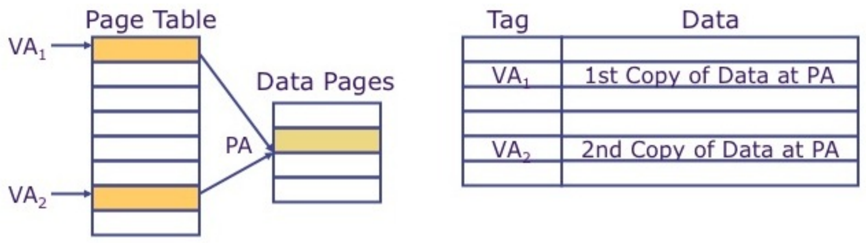


Figure 13.18: Aliasing in virtually-addressed caches

Second, we may have an *aliasing* problem due to sharing of pages. This is illustrated in Fig. 13.18 *i.e.,* if two different PTEs map to the same physical page, then a particular word in physical memory has two different virtual addresses, and therefore may have two different entries in a virtually-addressed cache. This is problematic because, for example, suppose the program writes via one of the virtual addresses, the other entry may not be marked dirty. Worse, it violates one of the normal properties and assumptions of memory, *i.e.,* that every memory location is independent, and that writes to one location do not magically change some other location.

The general solution to the aliasing problem is to forbid it—to never allow aliases to coexist in the cache (at least for addresses in writable pages). In early SPARCs, which had direct-mapped caches, this was enforced by making sure that VAs of shared pages agreed on cache index bits. This ensured that aliases always collided in the direct-mapped cache, so that only one could be in the cache.

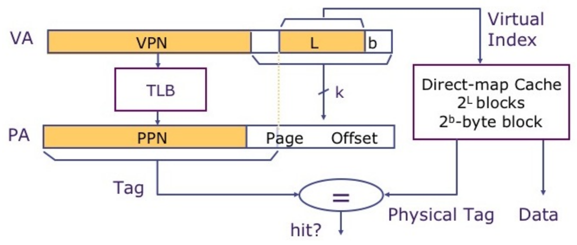


Figure 13.19: Concurrent access to the TLB and cache

Fig. 13.19 shows details of concurrent lookup in the TLB and cache. We use L bits outside the VPN bits to access the cache, since these are available before TLB lookup and invariant to the TLB translation. The cache entries contain tags based on physical addresses, which are compared with the output of TLB translation. Note that if the L field extends into the VPN field of the address, the cache becomes partially virtually addressed, which has to dealt with as discussed earlier (e.g., disambiguating with process ids).

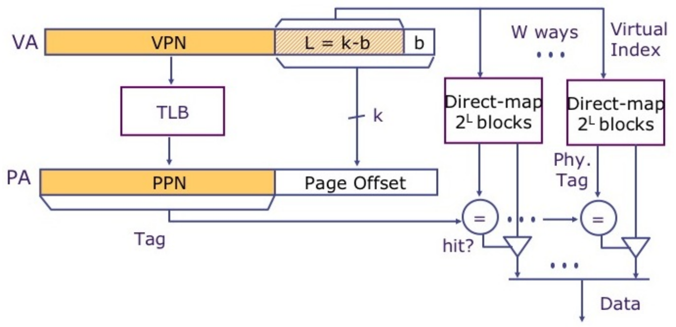


Figure 13.20: A cache with virtual index and physical tag

Fig. 13.20 shows a cache with a virtual index and physical tag. Virtual address bits are used for indexing the cache, but the cache entries contain physical addresses, which are compared against the post-TLB PPN for a hit. Of course, this can be done on direct-mapped as well as W-way set-associative caches.

13.11 Conclusion

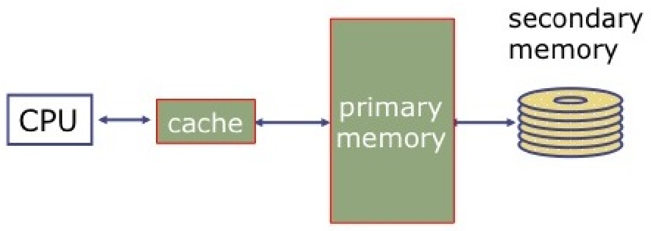


Figure 13.21: System memory hierarchy

Fig. 13.21 reprises the major components of the overall system memory hierarchy. If we were to add more detail, we would also include the register file in the processor at the top of the hierarchy, and possible multiple levels of cache, called Level 1 (L1), Level 2 (L2), Level 3 (L3) and so on. Modern processors often have 3 levels of cacheing or more. Many of the issues of cacheing are in principle the same at all levels of the memory hierarchy: associativity and lookup algorithms, clean vs. dirty, replacement policies, inclusiveness, aliasing, and so on. But the solutions are engineered differently because of their different scale (size), the size of units being cached, access patterns, and miss penalties. The table below summarizes some of the differences:

	Cacheing	Virtual Memory Demand Paging
Unit	cache line a.k.a. cache block	page
Typical size	32 bytes	4 Kbytes
Container	cache entry	page frame
Typical miss rates	1% to 20%	$< 0.001\%$
Typical hit latencies	1 clock cycle	100 cycles
Typical miss latencies	100 clock cycles	5 M cycles
Typical miss handling	in hardware	in software

Note for current draft of the book: In this chapter we have not shown any BSV code for virtual memory components. This is only because we do not yet have the corresponding software tools needed to exercise such solutions: operating system code using privileged instructions, TLB walkers, page fault handlers, etc. We expect all these to be available with a future edition of the book.

Chapter 14

Future Topics

This chapter lists a number of potential topics for future editions of this book.

14.1 Asynchronous Exceptions and Interrupts

14.2 Out-of-order pipelines

Register renaming, data flow. Precise exceptions.

14.3 Protection and System Issues

Protection issues and solutions. Virtualization. Capabilities.

14.4 I and D Cache Coherence

The coherence problem between Instruction and Data memories when Instruction memory is modifiable. Princeton architecture. Self-modifying code. On-the-fly optimization. JIT compiling. Binary translation.

14.5 Multicore and Multicore cache coherence

Multicore, SMPs.

Memory models and memory ordering semantics.

Multicore and directory-based cache coherence.

Synchronization primitives. Transactional memory.

14.6 Simultaneous Multithreading

SMT

14.7 Energy efficiency

Power and power management.

14.8 Hardware accelerators

Extending an ISA with HW-accelerated op codes, implemented in FPGAs or ASICs

Appendix A

SMIPS Reference

SMIPS ("Simple MIPS") is a subset of the full MIPS instruction set architecture (ISA). MIPS was one of the first commercial RISC (Reduced Instruction Set Computer) processors, and grew out of the earlier MIPS research project at Stanford University. MIPS originally stood for "Microprocessor without Interlocking Pipeline Stages" and the goal was to simplify the machine pipeline by requiring the compiler to schedule around pipeline hazards including a branch delay slot and a load delay slot (although those particular architectural choices have been abandoned). Today, MIPS CPUs are used in a wide range of devices: Casio builds handheld PDAs using MIPS CPUs, Sony uses two MIPS CPUs in the Playstation-2, many Cisco internet routers contain MIPS CPUs, and Silicon Graphics makes Origin supercomputers containing up to 512 MIPS processors sharing a common memory. MIPS implementations probably span the widest range for any commercial ISA, from simple single-issue in-order pipelines to quad-issue out-of-order superscalar processors.

There are several variants of the MIPS ISA. The ISA has evolved from the original 32-bit MIPS-I architecture used in the MIPS R2000 processor which appeared in 1986. The MIPS-II architecture added a few more instructions while retaining a 32-bit address space. The MIPS-II architecture also added hardware interlocks for the load delay slot. In practice, compilers couldn't fill enough of the load delay slots with useful work and the NOPs in the load delay slots wasted instruction cache space. (Removing the branch delay slots might also have been a good idea, but would have required a second set of branch instruction encodings to remain backwards compatible.) The MIPS-III architecture debuted with the MIPS R4000 processor, and this extended the address space to 64 bits while leaving the original 32-bit architecture as a proper subset. The MIPS-IV architecture was developed by Silicon Graphics to add many enhancements for floating-point computations and appeared first in the MIPS R8000 and later in the MIPS R10000. Over time, the MIPS architecture has been widely extended, occasionally in non-compatible ways, by different processor implementors. MIPS Technologies, the current owners of the architecture, are trying to rationalize the architecture into two broad groupings: MIPS32 is the 32-bit address space version, MIPS64 is the 64-bit address space version. There is also MIPS16, which is a compact encoding of MIPS32 that only uses 16 bits for each instruction. You can find a complete description of the MIPS instruction set at the MIPS Technologies web site [7] or in the book by Kane and Heinrich [8]. The book by Sweetman also explains MIPS programming [12]. Another source of MIPS details and implementation ideas is "Computer Organization and Design: The Hardware/Software Interface" [5].

Our subset, SMIPS, implements a subset of the MIPS32 ISA. It does not include floating point instructions, trap instructions, misaligned load/stores, branch and link instructions, or branch likely instructions. There are three SMIPS variants which are discussed in more detail in Appendix A. SMIPSv1 has only five instructions and it is mainly used as a toy ISA for instructional purposes. SMIPSv2 includes the basic integer, memory, and control instructions. It excludes multiply instructions, divide instructions, byte/halfword loads/stores, and instructions which cause arithmetic overflows. Neither SMIPSv1 or SMIPSv2 support exceptions, interrupts, or most of the system coprocessor. SMIPSv3 is the full SMIPS ISA and includes all the instructions in our MIPS subset.

A.1 Basic Architecture

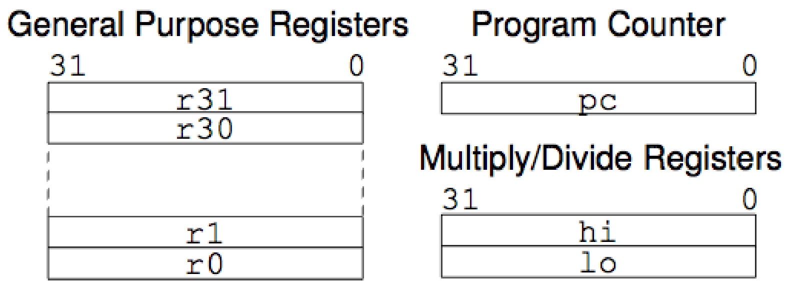


Figure A.1: SMIPS CPU Registers

Fig. A.1 shows the programmer visible state in the CPU. There are 31 general purpose 32-bit registers `r1`-`r31`. Register `r0` is hardwired to the constant 0. There are three special registers defined in the architecture: two registers `hi` and `lo` are used to hold the results of integer multiplies and divides, and the program counter `pc` holds the address of the instruction to be executed next. These special registers are used or modified implicitly by certain instructions.

SMIPS differs significantly from the MIPS32 ISA in one very important respect. SMIPS does *not* have a programmer-visible branch delay slot. Although this slightly complicates the control logic required in simple SMIPS pipelines, it greatly simplifies the design of more sophisticated out-of- order and superscalar processors. As in MIPS32, Loads are fully interlocked and thus there is no programmer-visible load delay slot.

Multiply instructions perform 32-bit $\times$ 32-bit $\rightarrow$ 64-bit signed or unsigned integer multiplies placing the result in the `hi` and `lo` registers. Divide instructions perform a 32-bit/32-bit signed or unsigned divide returning both a 32-bit integer quotient and a 32-bit remainder. Integer multiplies and divides can proceed in parallel with other instructions provided the hi and lo registers are not read.

The SMIPS CPU has two operating modes: user mode and kernel mode. The current operating mode is stored in the KUC bit in the system coprocessor (COP0) `status` register. The CPU normally operates in user mode until an exception forces a switch into kernel mode. The CPU will then normally execute an exception handler in kernel mode before executing a Return From Exception (ERET) instruction to return to user mode.

A.2 System Control Coprocessor (CP0)

The SMIPS system control coprocessor contains a number of registers used for exception handling, communication with a test rig, and the counter/timer. These registers are read and written using the MIPS standard MFC0 and MTC0 instructions respectively. User mode can access the system control coprocessor only if the `cu[0]` bit is set in the status register. Kernel mode can always access CP0, regardless of the setting of the `cu[0]` bit. CP0 control registers are listed in Table A.1.

Number	Register	Description
0-7		*unused*
8	badvaddr	Bad virtual address.
9	count	Counter/timer register.
10		*unused*
11	compare	Timer compare register.
12	status	Status register.
13	cause	Cause of last exception.
14	epc	Exception program counter.
15-19		*unused*
20	fromhost	Test input register.
21	tohost	Test output register.
22-31		*unused*

Table A.1: CP0 control registers

A.2.1 Test Communication Registers

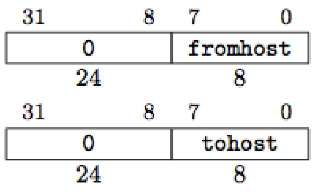


Figure A.2: Fromhost and Tohost Register Formats

There are two registers used for communicating and synchronizing with an external host test system. Typically, these will be accessed over a scan chain. The `fromhost` register is an 8-bit read only register that contains a value written by the host system. The `tohost` register is an 8-bit read/write register that contains a value that can be read back by the host system. The `tohost` register is cleared by reset to simplify synchronization with the host test rig. Their format is shown in Fig. A.2.

A.2.2 Counter/Timer Registers

SMIPS includes a counter/timer facility provided by the two coprocessor 0 registers `count` and `compare`. Both registers are 32 bits wide and are both readable and writeable. Their format is shown in Fig. A.3.

31 0
`count`
32

31 0
`compare`
32

Figure A.3: Count and Compare Registers.

The `count` register contains a value that increments once every clock cycle. The `count` register is normally only written for initialization and test purposes. A timer interrupt is flagged in `ip7` in the `cause` register when the `count` register reaches the same value as the `compare` register. The interrupt will only be taken if both `im7` and `iec` in the `status` register are set. The timer interrupt flag in `ip7` can only be cleared by writing the `compare` register. The `compare` register is usually only read for test purposes.

A.2.3 Exception Processing Registers

A number of CP0 registers are used for exception processing.

Status Register

31 28	27 16	15 8	7 6	5	4	3	2	1	0
CU	0	IM	0	KUo	IEo	KUp	IEp	KUc	IEc
4	12	8	2	1	1	1	1	1	1

Figure A.4: Status Register Format

The `status` register is a 32-bit read/write register formatted as shown in Fig. A.4. The `status` register keeps track of the processor's current operating state.

The CU field has a single bit for each coprocessor indicating if that coprocessor is usable. Bits 29-31, corresponding to coprocessors 1, 2, and 3, are permanently wired to 0 as these coprocessors are not available in SMIPS. Coprocessor 0 is always accessible in kernel mode regardless of the setting of bit 28 of the `status` register.

The IM field contains interrupt mask bits. Timer interrupts are disabled by clearing `im7` in bit 15. External interrupts are disabled by clearing `im6` in bit 14. The other bits within the IM field are not used on SMIPS and should be written with zeros. Table A.4 includes a listing of interrupt bit positions and descriptions.

The KUc/IEc/KUp/IEp/KUo/IEo bits form a three level stack holding the operating mode (ker- nel=0/user=1) and global interrupt enable (disabled=0/enabled=1) for the current state, and the two states before the two previous exceptions.

When an exception is taken, the stack is shifted left 2 bits and zero is written into KUc and IEc. When a Restore From Exception (RFE) instruction is executed, the stack is shifted right 2 bits, and the values in KUo/IEo are unchanged.

Cause Register

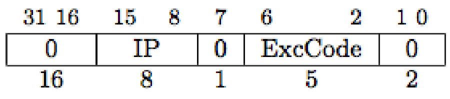


Figure A.5: Cause Register Format

The `cause` register is a 32-bit register formatted as shown in Fig. A.5. The `cause` register contains information about the type of the last exception and is read only.

The ExcCode field contains an exception type code. The values for ExcCode are listed in Table A.2. The ExcCode field will typically be masked off and used to index into a table of software exception handlers.

ExcCode	Mnemonic	Description
0	Hint	External interrupt
2	Tint	Timer interrupt
4	AdEL	Address or misalignment error on load
5	AdES	Address or misalignment error on store
6	AdEF	Address or misalignment error on fetch
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
12	Ov	Arithmetic Overflow

Table A.2: Exception Types

If the Branch Delay bit (BD) is set, the instruction that caused the exception was executing in a branch delay slot and epc points to the immediately preceding branch instruction. Otherwise,

The IP field indicates which interrupts are pending. Field `ip7` in bit 15 flags a timer interrupt. Field ip6 in bit 14 flags an external interrupt from the host test setup. The other IP bits are unused in SMIPS and should be ignored when read. Table 4 includes a listing of interrupt bit positions and descriptions.

Exception Program Counter

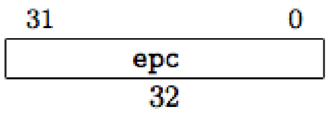


Figure A.6: EPC Register

`Epc` is a 32-bit read only register formatted as shown in Fig. A.6. When an exception occurs, `epc` is written with the virtual address of the instruction that caused the exception.

Bad Virtual Address

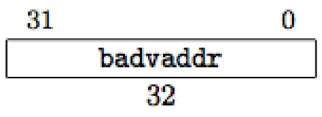


Figure A.7: Badvaddr Register

`Badvaddr` is a 32-bit read only register formatted as shown in Fig. A.7. When a memory address error generates an AdEL or AdES exception, `badvaddr` is written with the faulting virtual address. The value in `badvaddr` is undefined for other exceptions.

A.3 Addressing and Memory Protection

SMIPS has a full 32-bit virtual address space with a full 32-bit physical address space. Sub-word data addressing is big-endian on SMIPS.

The virtual address space is split into two 2 GB segments, a kernel only segment (*kseg*) from `0x0000 0000` to `0x7fff ffff`, and a kernel and user segment (*kuseg*) from `0x8000 0000` to `0xffff ffff`. The segments are shown in Fig. A.8.

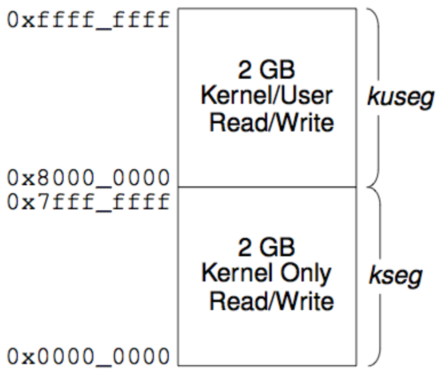


Figure A.8: SMIPS virtual address space

In kernel mode, the processor can access any address in the entire 4 GB virtual address space. In user mode, instruction fetches or scalar data accesses to the kseg segment are illegal and cause a synchronous exception. The AdEF exception is generated for an illegal instruction fetch, and AdEL and AdES exceptions are generated for illegal loads and stores respectively. For faulting stores, no data memory will be written at the faulting address.

There is no memory translation hardware on SMIPS. Virtual addresses are directly used as physical addresses in the external memory system. The memory controller simply ignores unused high order address bits, in which case each physical memory address will be shadowed multiple times in the virtual address space.

A.4 Reset, Interrupt, and Exception Processing

There are three possible sources of disruption to normal program flow: reset, interrupts (asyn- chronous exceptions), and synchronous exceptions. Reset and interrupts occur asynchronously to the executing program and can be considered to occur *between* instructions. Synchronous exceptions occur *during* execution of a particular instruction.

If more than one of these classes of event occurs on a given cycle, reset has highest priority, and all interrupts have priority over all synchronous exceptions. The tables below show the priorities of different types of interrupt and synchronous exception.

The flow of control is transferred to one of two locations as shown in Table A.3. Reset has a separate vector from all other exceptions and interrupts.

Vector Address	Cause
`0x0000_1000`	Reset
`0x0000_1100`	Exceptions and internal interrupts

Table A.3: SMIPS Reset, Exception, and Interrupt Vectors.

A.4.1 Reset

When the external reset is deasserted, the PC is reset to `0x0000 1000`, with `kuc` set to 0, and `iec` set to 0. The effect is to start execution at the reset vector in kernel mode with interrupts disabled. The `tohost` register is also set to zero to allow synchronization with the host system. All other state is undefined.

A typical reset sequence is shown in Fig. A.9.

```
reset_vector:
    mtc0 zero, $9     # Initialize counter.
    mtc0 zero, $11    # Clear any timer interrupt in compare.

    # Initialize status with desired CU, IM, and KU/IE fields.
    li k0, (CU_VAL|IM_VAL|KUIE_VAL)
    mtc0 k0, $12      # Write to status register.

    j kernel_init     # Initialize kernel software.
```

Figure A.9: SMIPS virtual address space

A.4.2 Interrupts

The two interrupts possible on SMIPS are listed in Table A.4 in order of decreasing priority.

Vector	ExcCode	Mnemonic	IM/IP Index	Description
Highest Priority				
`0x0000_1100`	0	Hint	6	Tester interrupt.
`0x0000_1100`	2	Tint	7	Timer interrupt.
Lowest Priority				

Table A.4: SMIPS Interrupts.

All SMIPS interrupts are level triggered. For each interrupt there is an IP flag in the `cause` register that is set if that interrupt is pending, and an IM flag in the `status` register that enables the interrupt when set. In addition there is a single global interrupt enable bit, `iec`, that disables all interrupts if cleared. A particular interrupt can only occur if both IP and IM for that interrupt are set and `iec` is set, and there are no higher priority interrupts.

The host external interrupt flag IP6 can be written by the host test system over a scan interface. Usually a protocol over the host scan interface informs the host that it can clear down the interrupt flag.

The timer interrupt flag IP7 is set when the value in the `count` register matches the value in the `compare` register. The flag can only be cleared as a side-effect of a MTC0 write to the `compare` register.

When an interrupt is taken, the PC is set to the interrupt vector, and the KU/IE stack in the `status` register is pushed two bits to the left, with KUc and IEc both cleared to 0. This starts the interrupt handler running in kernel mode with further interrupts disabled. The `exccode` field in the `cause` register is set to indicate the type of interrupt.

The `epc` register is loaded with a restart address. The `epc` address can be used to restart execution after servicing the interrupt.

A.4.3 Synchronous Exceptions

Synchronous exceptions are listed in Table A.5 in order of decreasing priority.

ExcCode	Mnemonic	Description
		Highest Priority
6	AdEF	Address or misalignment error on fetch.
10	RI	Reserved instruction exception.
8	Sys	Syscall exception.
9	Bp	Breakpoint exception.
12	Ov	Arithmetic Overflow.
4	AdEL	Address or misalignment error on load.
5	AdES	Address or misalignment error on store.
		Lowest Priority

Table A.5: SMIPS Synchronous Exceptions.

After a synchronous exception, the PC is set to `0x0000\_1100`. The stack of kernel/user and interrupt enable bits held in the `status` register is pushed left two bits, and both `kuc` and `iec` are set to 0.

The `epc` register is set to point to the instruction that caused the exception. The `exccode` field in the cause register is set to indicate the type of exception.

If the exception was a coprocessor unusable exception (CpU), the `ce` field in the `cause` register is set to the coprocessor number that caused the error. This field is undefined for other exceptions.

The overflow exception (Ov) can only occur for ADDI, ADD, and SUB instructions.

If the exception was an address error on a load or store (AdEL/AdES), the `badvaddr` register is set to the faulting address. The value in `badvaddr` is undefined for other exceptions.

All unimplemented and illegal instructions should cause a reserved instruction exception (RI).

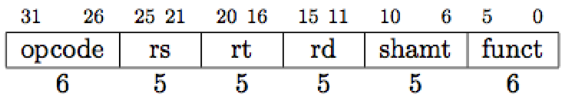
A.5 Instruction Semantics and Encodings

SMIPS uses the standard MIPS instruction set.

A.5.1 Instruction Formats

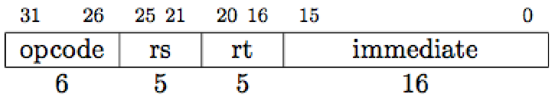
There are three basic instruction formats, R-type, I-type, and J-type. These are a fixed 32 bits in length, and must be aligned on a four-byte boundary in memory. An address error exception (AdEF) is generated if the PC is misaligned on an instruction fetch.

R-Type



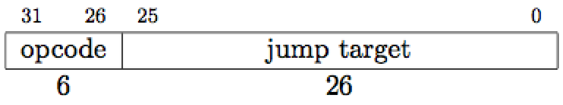
R-type instructions specify two source registers (*rs* and *rt*) and a destination register (*rd*). The 5-bit *shamt* field is used to specify shift immediate amounts and the 6-bit *funct* code is a second opcode field.

I-Type



I-type instructions specify one source register (*rs*) and a destination register (*rt*). The second source operand is a sign or zero-extended 16-bit immediate. Logical immediate operations use a zero-extended immediate, while all others use a sign-extended immediate.

J-Type



J-type instructions encode a 26-bit jump target address. This value is shifted left two bits to give a byte address then combined with the top four bits of the current program counter.

A.5.2 Instruction Categories

MIPS instructions can be grouped into several basic categories: loads and stores, computation instructions, branch and jump instructions, and coprocessor instructions.

Load and Store Instructions

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16
LB/LH/LW/LBU/LHU/LL	base	dest	offset
SB/SH/SW/SC	base	src	offset

Load and store instructions transfer a value between the registers and memory and are encoded with the I-type format. The effective address is obtained by adding register *rs* to the sign-extended immediate. Loads place a value in register *rt*. Stores write the value in register *rt* to memory.

The LW and SW instructions load and store 32-bit register values respectively. The LH instruction loads a 16-bit value from memory and sign extends this to 32-bits before storing into register *rt*. The LHU instruction zero-extends the 16-bit memory value. Similarly LB and LBU load sign and zero-extended 8-bit values into register *rt* respectively. The SH instruction writes the low-order 16 bits of register *rt* to memory, while SB writes the low-order 8 bits.

The effective address must be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit loads/stores and a two-byte boundary for 16-bit loads/store). If not, an address exception (AdEL/AdES) is generated.

The load linked (LL) and store conditional (SC) instructions are used as primitives to implement atomic read-modify-write operations for multiprocessor synchronization. The LL instruction performs a standard load from the effective address (base+offset), but as a side effect the instruction should set a programmer invisible link address register. If for any reason atomicity is violated, then the link address register will be cleared. When the processor executes the SC instruction first, it first verifies that the link address register is still valid. If link address register is valid then the SC executes as a standard SW instruction except that the *src* register is overwritten with a one to indicate success. If the link address register is invalid, the then SW instruction overwrites the *src* register with a zero to indicate failure. There are several reasons why atomicity might be violated. If the processor takes an exception after an LL instruction but before the corresponding SC instruction is executed then the link address register will be cleared. In a multi-processor system, if a different processor uses a SC instruction to write the same location then the link address register will also be cleared.

Computational Instructions

Computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register *rt* for register-immediate instructions and *rd* for register-register instructions. There are only eight register-immediate computational instructions.

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16
ADDI/ADDIU/SLTI/SLTIU	src	dest	sign-extended immediate
ANDI/ORI/XORI/LUI	src	dest	zero-extended immediate

ADDI and ADDIU add the sign-extended 16-bit immediate to register *rs*. The only difference between ADD and ADDIU is that ADDI generates an arithmetic overflow exception if the signed result would overflow 32 bits. SLTI (set less than immediate) places a 1 in the register *rt* if register *rs* is strictly less than the sign-extended immediate when both are treated as signed 32-bit numbers, else a 0 is written to *rt*. SLTIU is similar but compares the values as unsigned 32-bit numbers. [**NOTE: Both ADDIU and SLTIU sign-extend the immediate, even though they operate on unsigned numbers.**]

ANDI, ORI, XORI are logical operations that perform bit-wise AND, OR, and XOR on register *rs* and the zero-extended 16-bit immediate and place the result in *rt*.

LUI (load upper immediate) is used to build 32-bit immediates. It shifts the 16-bit immediate into the high-order 16-bits, shifting in 16 zeros in the low order bits, then places the result in register *rt*. The *rs* field must be zero.

[**NOTE: Shifts by immediate values are encoded in the R-type format using the *shamt* field.**]

Arithmetic R-type operations are encoded with a zero value (SPECIAL) in the major opcode. All operations read the *rs* and *rt* registers as source operands and write the result into register *rd*. The 6-bit *funct* field selects the operation type from ADD, ADDU, SUB, SUBU, SLT, SLTU, AND, OR, XOR, and NOR.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	src1	src2	dest	0	ADD/ADDU/SUB/SUBU
SPECIAL=0	src1	src2	dest	0	SLT/SLTU
SPECIAL=0	src1	src2	dest	0	AND/OR/XOR/NOR

ADD and SUB perform add and subtract respectively, but signal an arithmetic overflow if the result would overflow the signed 32-bit destination. ADDU and SUBU are identical to ADD/SUB except no trap is created on overflow. SLT and SLTU performed signed and unsigned compares respectively, writing 1 to *rd* if *rs* < *rt*, 0 otherwise. AND, OR, XOR, and NOR perform bitwise logical operations. [**NOTE: NOR *rd*, *rx*, *rx* performs a logical inversion (NOT) of register *rx*.**]

Shift instructions are also encoded using R-type instructions with the SPECIAL major opcode. The operand that is shifted is always register *rt*. Shifts by constant values

(SLL/SRL/SRA) have the shift amount encoded in the *shamt* field. Shifts by variable values (SLLV/SRLV/SRAV) take the shift amount from the bottom five bits of register *rs*. SLL/SLLV are logical left shifts, with zeros shifted into the least significant bits. SRL/SRLV are logical right shifts with zeros shifted into the most significant bits. SRA/SRAV are arithmetic right shifts which shift in copies of the original sign bit into the most significant bits.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	0	src	dest	shift	SLL/SRL/SRA
SPECIAL=0	shift	src	dest	0	SLLV/SRLV/SRAV

Multiply and divide instructions target the `hi` and `lo` registers and are encoded as R-type instructions under the SPECIAL major opcode. These instructions are fully interlocked in hardware. Multiply instructions take two 32-bit operands in registers *rs* and *rt* and store their 64-bit product in registers `hi` and `lo`. MULT performs a signed multiplication while MULTU performs an unsigned multiplication. DIV and DIVU perform signed and unsigned divides of register *rs* by register *rt* placing the quotient in `lo` and the remainder in `hi`. Divides by zero do not cause a trap. A software check can be inserted if required.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	src1	src2	0	0	MULT/MULTU/DIV/DIVU
SPECIAL=0	0	0	dest	0	MFHI/MFLO
SPECIAL=0	src	0	0	0	MTHI/MTLO

The values calculated by a multiply or divide instruction are retrieved from the `hi` and `lo` registers using the MFHI (move from `hi`) and MFLO (move from `lo`) instructions, which write register *rd*. MTHI (move to `hi`) and MTLO (move to `lo`) instructions are also provided to allow the multiply registers to be written with the value in register *rs* (these instructions are used to restore user state after a context swap).

Jump and Branch Instructions

Jumps and branches can change the control flow of a program. Unlike the MIPS32 ISA, the SMIPS ISA does not have a programmer visible branch delay slot.

Absolute jumps (J) and jump and link (JAL) instructions use the J-type format. The 26-bit jump target is concatenated to the high order four bits of the program counter of the delay slot, then shifted left two bits to form the jump target address (using Verilog notation, the target address is `{pc_plus4[31:28],target[25:0],2'b0}`. JAL stores the address of the instruction following the jump (PC+4) into register `r31`.

31 26	25 0
opcode	jump target
6	26
J/JAL	offset

The indirect jump instructions, JR (jump register) and JALR (jump and link register), use the R-type encoding under a SPECIAL major opcode and jump to the address contained in register *rs*. JALR writes the link address into register *rd*.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
SPECIAL=0	src	0	0	0	JR
SPECIAL=0	src	0	dest	0	JALR

All branch instructions use the I-type encoding. The 16-bit immediate is sign-extended, shifted left two bits, then added to the address of the instruction in the delay slot (PC+4) to give the branch target address.

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16
BEQ/BNE	src1	src2	offset
BLEZ/BGTZ	src	0	offset
REGIMM	src	BLTZ/BGEZ	offset

BEQ and BNE compare two registers and take the branch if they are equal or unequal respectively. BLEZ and BGTZ compare one register against zero, and branch if it is less than or equal to zero, or greater than zero, respectively. BLTZ and BGEZ examine the sign bit of the register *rs* and branch if it is negative or positive respectively.

System Coprocessor Instructions

The MTC0 and MFCO instructions access the control registers in coprocessor 0, transferring a value from/to the coprocessor register specified in the *rd* field to/from the CPU register specified in the *rt* field. It is important to note that the coprocessor register is always in the *rd* field and the CPU register is always in the *rt* field regardless of which register is the source and which is the destination.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
COP0	MF	dest	cop0src	0	0
COP0	MT	src	cop0dest	0	0
COP0	CO	0	0	0	ERET

The restore from exception instruction, ERET, returns to the interrupted instruction at the completion of interrupt or exception processing. An ERET instruction should pop the top value of the interrupt and kernel/user status register stack, restoring the previous values.

Coprocessor 2 Instructions

Coprocessor 2 is reserved for an implementation defined hardware unit. The MTC2 and MFC2 instructions access the registers in coprocessor 2, transferring a value from/to the coprocessor register specified in the *rd* field to/from the CPU register specified in the *rt* field. The CTC2 and CFC2 instructions serve a similar process. The Coprocessor 2 implementation is free to handle the coprocessor register specifiers in MTC2/MFC2 and CTC2/CFC2 in any way it wishes.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6
COP2	MF	dest	cop2src	0	0
COP2	MT	src	cop2dest	0	0
COP2	CF	dest	cop2src	0	0
COP2	CT	src	cop2dest	0	0

The LWC2 and SWC2 instructions transfer values between memory and the coprocessor registers. Note that although *cop2dest* and *cop2src* fields are coprocessor register specifiers, the ISA does not define how these correspond to the coprocessor register specifiers in other Coprocessor 2 instructions.

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16
LWC2	base	cop2dest	offset
SWC2	base	cop2src	offset

The COP2 instruction is the primary mechanism by which a programmer can specify instruction bits to control Coprocessor 2. The 25-bit *copfunc* field is compeletely implementation dependent.

31 26	25 25	24 0
opcode		coprocessor function
6	1	25
COP2	C0	copfunc

Special Instructions

The SYSCALL and BREAK instructions are useful for implementing operating systems and debuggers. The SYNC instruction can be necessary to guarantee strong load/store ordering in modern multi-processors with sophisticated memory systems.

The SYSCALL and BREAK instructions cause and immediate syscall or break exception. To access the code field for use as a software parameter, the exception handler must load the memory location containing the syscall instruction.

31 26	25 6	5 0
opcode	0	funct
6	20	6
SPECIAL=0	code	SYSCALL
SPECIAL=0	code	BREAK
SPECIAL=0	0	SYNC

The SYNC instruction is used to order loads and stores. All loads and stores before the SYNC instruction must be visible to all other processors in the system before any of the loads and stores following the SYNC instruction are visible.

A.5.3 SMIPS Variants

The SMIPS specification defines three SMIPS subsets: SMIPSv1, SMIPSv2, and SMIPSv3. SMIPSv1 includes the following five instructions: ADDIU, BNE, LW, SW, and MTC0. The `tohost` register is the only implemented system coprocessor register. SMIPSv2 includes all of the simple arithmetic instructions except for those which throw overflow exceptions. It does not include multiply or divide instructions. SMIPSv2 only supports word loads and stores. All jumps and branches are supported. Neither SMIPSv1 or SMIPSv2 support exceptions, interrupts, or most of the system coprocessor. SMIPSv3 is the full SMIPS ISA and includes everything described in this document except for Coprocessor 2 instructions. Table A.7 notes which instructions are supported in each variant.

A.5.4 Unimplemented Instructions

Several instructions in the MIPS32 instruction set are not supported by the SMIPS. These instructions should cause a reserved instruction exception (RI) and can be emulated in software by an exception handler.

The misaligned load/store instructions, Load Word Left (LWL), Load Word Right (LWR), Store Word Left (SWL), and Store Word Right (SWR), are not implemented. A trap handler can emulate the misaligned access. Compilers for SMIPS should avoid generating these instructions, and should instead generate code to perform the misaligned access using multiple aligned accesses.

The MIPS32 trap instructions, TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, TNEI, are not implemented. The illegal instruction trap handler can perform the comparison and if the condition is met jump to the appropriate exception routine, otherwise resume user mode execution after the trap instruction. Alternatively, these instructions may be synthesized by the assembler, or simply avoided by the compiler.

The floating point coprocessor (COP1) is not supported. All MIPS32 coprocessor 1 instructions are trapped to allow emulation of floating-point. For higher performance, compilers for SMIPS could directly generate calls to software floating point code libraries rather than emit coprocessor instructions that will cause traps, though this will require modifying the standard MIPS calling convention.

Branch likely and branch and link instructions are not implemented and cannot be emulated so they should be avoided by compilers for SMIPS.

A.6 Instruction listing for SMIPS

31 26	25 21	20 16	15 11	10 6	5 0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type

Load and Store Instructions

v3	100000	base	dest	signed offset	LB rt, offset(rs)
v3	100001	base	dest	signed offset	LH rt, offset(rs)
v1	100011	base	dest	signed offset	LW rt, offset(rs)
v3	100100	base	dest	signed offset	LBU rt, offset(rs)
v3	100101	base	dest	signed offset	LHU rt, offset(rs)
v3	110000	base	dest	signed offset	LL rt, offset(rs)
v3	101000	base	src	signed offset	SB rt, offset(rs)
v3	101001	base	src	signed offset	SH rt, offset(rs)
v1	101011	base	src	signed offset	SW rt, offset(rs)
v3	111000	base	src	signed offset	SC rt, offset(rs)

I-Type Computational Instructions

v3	001000	src	dest	signed immediate	ADDI rt, rs, signed-imm.
v1	001001	src	dest	signed immediate	ADDIU rt, rs, signed-imm.
v2	001010	src	dest	signed immediate	SLTI rt, rs, signed-imm.
v2	001011	src	dest	signed immediate	SLTIU rt, rs, signed-imm.
v2	001100	src	dest	zero-ext. immediate	ANDI rt, rs, zero-ext-imm.
v2	001101	src	dest	zero-ext. immediate	ORI rt, rs, zero-ext-imm.
v2	001110	src	dest	zero-ext. immediate	XORI rt, rs, zero-ext-imm.
v2	001111	00000	dest	zero-ext. immediate	LUI rt, zero-ext-imm.

R-Type Computational Instructions

v2	000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
v2	000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
v2	000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
v2	000000	rshamt	src	dest	00000	000100	SLLV rd, rt, shamt
v2	000000	rshamt	src	dest	00000	000110	SRLV rd, rt, shamt
v2	000000	rshamt	src	dest	00000	000111	SRAV rd, rt, shamt
v3	000000	src1	src2	dest	00000	100000	ADD rd, rt, shamt
v2	000000	src1	src2	dest	00000	100001	ADDU rd, rt, shamt
v3	000000	src1	src2	dest	00000	100010	SUB rd, rt, shamt
v2	000000	src1	src2	dest	00000	100011	SUBU rd, rt, shamt
v2	000000	src1	src2	dest	00000	100100	AND rd, rt, shamt
v2	000000	src1	src2	dest	00000	100101	OR rd, rt, shamt
v2	000000	src1	src2	dest	00000	100110	XOR rd, rt, shamt
v2	000000	src1	src2	dest	00000	100111	NOR rd, rt, shamt
v2	000000	src1	src2	dest	00000	101010	SLT rd, rt, shamt
v2	000000	src1	src2	dest	00000	101011	SLTU rd, rt, shamt

Multiply/Divide Instructions

v3	000000	00000	00000	dest	00000	010000	MFHI rd
v3	000000	rs	00000	00000	00000	010001	MTHI rs
v3	000000	00000	00000	dest	00000	010010	MFLO rd
v3	000000	rs	00000	00000	00000	010011	MTLO rs
v3	000000	src1	src2	00000	00000	011000	MULT rs, rt
v3	000000	src1	src2	00000	00000	011001	MULTU rs, rt
v3	000000	src1	src2	00000	00000	011010	DIV rs, rt
v3	000000	src1	src2	00000	00000	011011	DIVU rs, rt

Jump and Branch Instructions

v2	000010	target					J target
v2	000011	target					JAL target
v2	000000	src	00000	00000	00000	001000	JR rs
v2	000000	src	00000	dest	00000	001001	JALR rd, rs
v2	000100	src1	src2	signed offset			BEQ rs, rt, offset
v2	000101	src1	src2	signed offset			BNE rs, rt, offset
v2	000110	src	00000	signed offset			BLEZ rs, rt, offset
v2	000111	src	00000	signed offset			BGTZ rs, rt, offset
v2	000001	src	00000	signed offset			BLTZ rs, rt, offset
v2	000001	src	00001	signed offset			BGEZ rs, rt, offset

System Coprocessor (COP0) Instructions

v2	010000	00000	dest	cop0src	00000	000000	MFC0 rt, rd
v2	010000	00100	src	cop0dest	00000	000000	MTC0 rt, rd
v3	010000	10000	00000	00000	00000	011000	ERET

Coprocessor 2 (COP2) Instructions

–	010010	00000	dest	cop2src	00000	000000	MFC2 rt, rd
–	010010	00100	src	cop2dest	00000	000000	MTC2 rt, rd
–	010010	00010	dest	cop2src	00000	000000	CFC2 rt, rd
–	010010	00110	src	cop2dest	00000	000000	CTC2 rt, rd
–	110010	base	cop2dest	signed offset			LWC2 rt, offset(rs)
–	111010	base	cop2src	signed offset			SWC2 rt, offset(rs)
–	010010	1				copfunc	COP2 copfunc

Special Instructions

v3	000000	00000	00000	00000	00000	001100	SYSCALL
v3	000000	00000	00000	00000	00000	001101	BREAK
v3	000000	00000	00000	00000	00000	001111	SYNC

Bibliography

[1] Bluespec, Inc. Bluespec$^{\text{TM}}$ SystemVerilog Version Reference Guide, 2010.

[2] Bluespec, Inc. Bluespec$^{\text{TM}}$ SystemVerilog Version User Guide, 2010.

[3] Bluespec, Inc. High-level "plug-and-play" specification, modeling and synthesis of pipelined architectures with Bluespec's PAClib, 2010.

[4] N. Dave, M. Pellauer, S. Gerding, and Arvind. 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In *Proc. 6th ACM/IEEE Intl. Conf. on Formal Methods and Models for Codesign (MEMOCODE), Napa Valley, CA*, July 2006.

[5] J. L. Hennessy and D. Patterson. *Computer Organization and Design: The Hardware/Software Interface (Second Edition).* Morgan Kaufmann, February 1997. ISBN 1558604286.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (Fifth Edition).* Morgan Kaufman, September 2011.

[7] M. T. Inc. *MIPS32 Architecture for Programmers.* 2002. http://www.mips.com/publications/processor_architecture.html.

[8] G. Kane and J. Heinrich. *MIPS RISC Architecture (2nd edition).* Prentice Hall, September 1991. ISBN 0135904722.

[9] R. S. Nikhil and K. R. Czeck. *BSV by Example.* CreateSpace, December 2010. ISBN-10: 1456418467; ISBN-13: 978-1456418465; avail: Amazon.com.

[10] D. L. Rosenband. The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. In *Proc. MEMOCODE'04*, June 2004.

[11] D. L. Rosenband. *A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 2005. Also: Memo-491, http://csg.csail.mit.edu/pubs/publications.html.

[12] D. Sweetman. *See MIPS Run.* Morgan Kaufmann, April 1999. ISBN 1558604103.