

HDL Example 7.15 MIPS INSTRUCTION MEMORY

Verilog

```
module imem(input [5:0] a,
            output [31:0] rd);

    reg [31:0] RAM[63:0];

    initial
    begin
        $readmemh ("memfile.dat",RAM);
    end

    assign rd = RAM[a]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;

entity imem is -- instruction memory
    port(a: in STD_LOGIC_VECTOR(5 downto 0);
         rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
    process is
        file mem_file: TEXT;
        variable L: line;
        variable ch: character;
        variable index, result: integer;
        type ramtype is array(63 downto 0) of STD_LOGIC_VECTOR
            (31 downto 0);
        variable mem: ramtype;
    begin
        -- initialize memory from file
        for i in 0 to 63 loop -- set all contents low
            mem(conv_integer(i)) := CONV_STD_LOGIC_VECTOR(0, 32);
        end loop;
        index := 0;
        FILE_OPEN(mem_file, "C:/mips/memfile.dat", READ_MODE);
        while not endfile(mem_file) loop
            readline(mem_file, L);
            result := 0;
            for i in 1 to 8 loop
                read(L, ch);
                if '0' <= ch and ch <= '9' then
                    result := result*16 + character'pos(ch) -
                        character'pos('0');
                elsif 'a' <= ch and ch <= 'f' then
                    result := result*16 + character'pos(ch) -
                        character'pos('a') + 10;
                else report "Format error on line" & integer'image
                    (index) severity error;
                end if;
            end loop;
            mem(index) := CONV_STD_LOGIC_VECTOR(result, 32);
            index := index + 1;
        end loop;

        -- read memory
        loop
            rd <= mem(CONV_INTEGER(a));
            wait on a;
        end loop;
    end process;
end;
```

7.7 EXCEPTIONS*

Section 6.7.2 introduced exceptions, which cause unplanned changes in the flow of a program. In this section, we enhance the multicycle processor to support two types of exceptions: undefined instructions

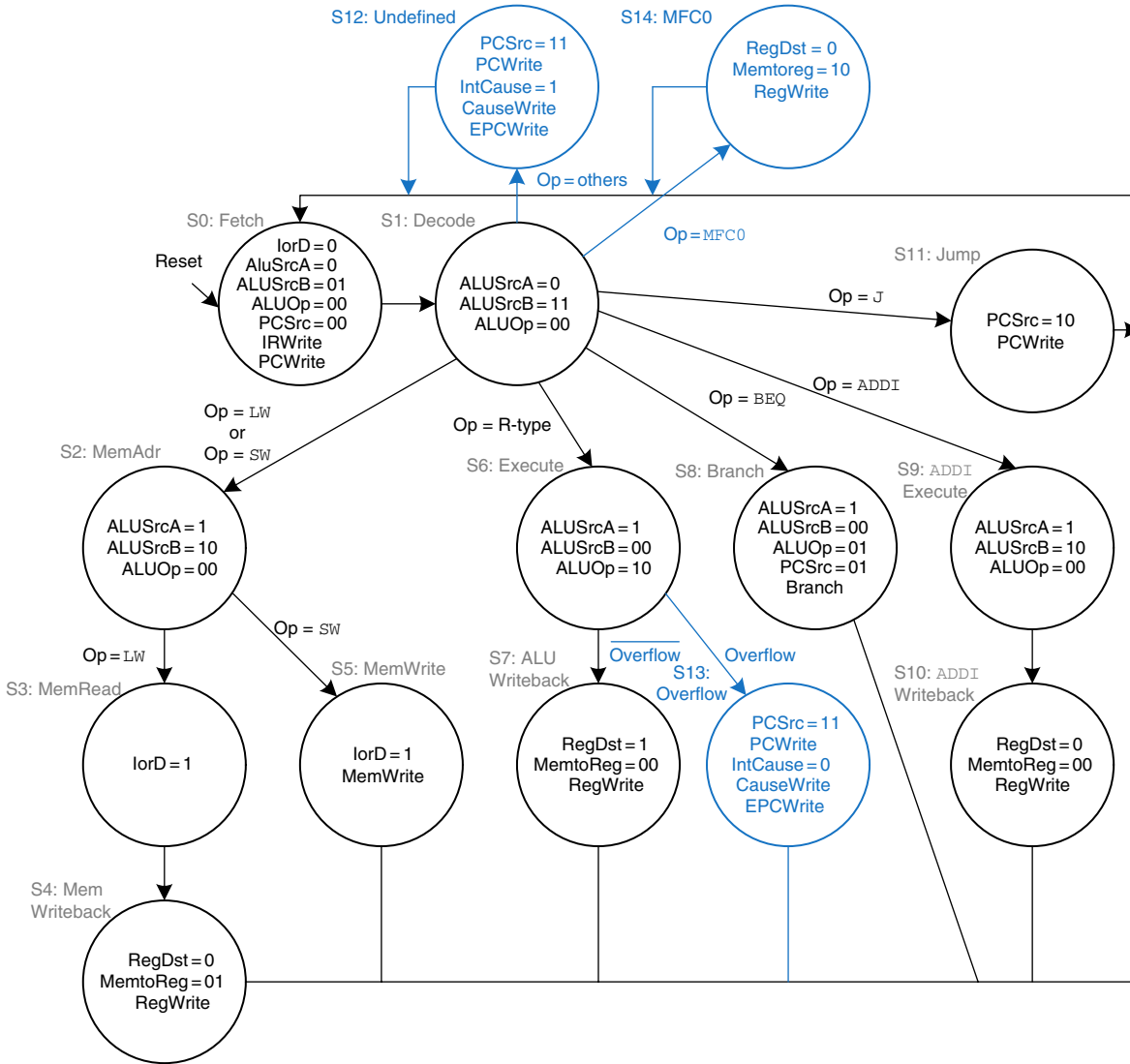


Figure 7.64 Controller supporting exceptions and mfc0

in the Cause register, and jumps to the exception handler. Note that, when an exception occurs, the instruction is discarded and the register file is not written. When a mfc0 instruction is decoded, the processor goes to S14 and writes the appropriate Coprocessor 0 register to the main register file.

7.8 ADVANCED MICROARCHITECTURE*

High-performance microprocessors use a wide variety of techniques to run programs faster. Recall that the time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction (CPI). Thus, to increase performance we would like to speed up the clock and/or reduce the CPI. This section surveys some existing speedup techniques. The implementation details become quite complex, so we will focus on the concepts. Hennessy & Patterson's *Computer Architecture* text is a definitive reference if you want to fully understand the details.

Every 2 to 3 years, advances in CMOS manufacturing reduce transistor dimensions by 30% in each direction, doubling the number of transistors that can fit on a chip. A manufacturing process is characterized by its *feature size*, which indicates the smallest transistor that can be reliably built. Smaller transistors are faster and generally consume less power. Thus, even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster. Moreover, smaller transistors enable placing more transistors on a chip. Microarchitects use the additional transistors to build more complicated processors or to put more processors on a chip. Unfortunately, power consumption increases with the number of transistors and the speed at which they operate (see Section 1.8). Power consumption is now an essential concern. Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors in some of the most complex systems that humans have ever built.

7.8.1 Deep Pipelines

Aside from advances in manufacturing, the easiest way to speed up the clock is to chop the pipeline into more stages. Each stage contains less logic, so it can run faster. This chapter has considered a classic five-stage pipeline, but 10 to 20 stages are now commonly used.

The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead, and cost. Longer pipelines introduce more dependencies. Some of the dependencies can be solved by forwarding, but others require stalls, which increase the CPI. The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew). This sequencing overhead makes adding more pipeline stages give diminishing returns. Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

Example 7.11 DEEP PIPELINES

Consider building a pipelined processor by chopping up the single-cycle processor into N stages ($N \geq 5$). The single-cycle processor has a propagation delay of 900 ps through the combinational logic. The sequencing overhead of a register is 50 ps. Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay. The five-stage pipeline in Example 7.9 has a CPI of 1.15. Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards. How many pipeline stages should be used to make the processor execute programs as fast as possible?

Solution: If the 900-ps combinational logic delay is divided into N stages and each stage also pays 50 ps of sequencing overhead for its pipeline register, the cycle time is $T_c = 900/N + 50$. The CPI is $1.15 + 0.1(N - 5)$. The time per instruction, or instruction time, is the product of the cycle time and the CPI. Figure 7.65 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 231 ps at $N = 11$ stages. This minimum is only slightly better than the 250 ps per instruction achieved with a six-stage pipeline.

In the late 1990s and early 2000s, microprocessors were marketed largely based on clock frequency ($1/T_c$). This pushed microprocessors to use very deep pipelines (20 to 31 stages on the Pentium 4) to maximize the clock frequency, even if the benefits for overall performance were questionable. Power is proportional to clock frequency and also increases with the number of pipeline registers, so now that power consumption is so important, pipeline depths are decreasing.

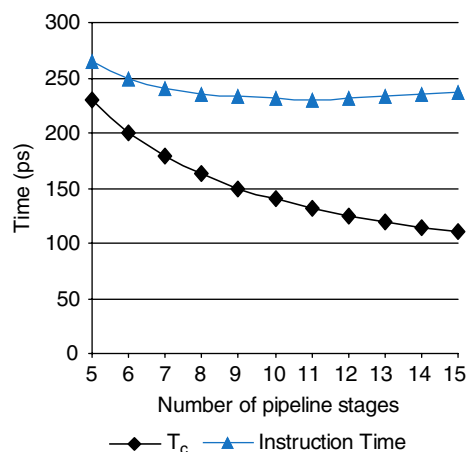


Figure 7.65 Cycle time and instruction time versus the number of pipeline stages

7.8.2 Branch Prediction

An ideal pipelined processor would have a CPI of 1. The branch misprediction penalty is a major reason for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger, because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a *branch predictor* to guess whether the branch should be taken. Recall that our pipeline from Section 7.5.3 simply predicted that branches are never taken.

Some branches occur when a program reaches the end of a loop (e.g., a `for` or `while` statement) and branches back to repeat the loop. Loops tend to be executed many times, so these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches should be taken. This is called *static branch prediction*, because it does not depend on the history of the program.

Forward branches are difficult to predict without knowing more about the specific program. Therefore, most processors use *dynamic branch predictors*, which use the history of program execution to guess whether a branch should be taken. Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table, sometimes called a *branch target buffer*, includes the destination of the branch and a history of whether the branch was taken.

To see the operation of dynamic branch predictors, consider the following loop code from Code Example 6.20. The loop repeats 10 times, and the `beq` out of the loop is taken only on the last time.

```
add $s1, $0, $0      # sum = 0
add $s0, $0, $0      # i = 0
addi $t0, $0, 10     # $t0 = 10

for:
    beq $s0, $t0, done # if i == 10, branch to done
    add $s1, $s1, $s0  # sum = sum + i
    addi $s0, $s0, 1   # increment i
    j    for
done:
```

A *one-bit dynamic branch predictor* remembers whether the branch was taken the last time and predicts that it will do the same thing the next time. While the loop is repeating, it remembers that the `beq` was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch does get taken. Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore,

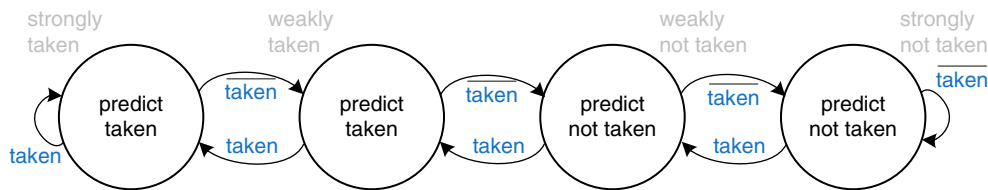


Figure 7.66 2-bit branch predictor state transition diagram

it incorrectly predicts that the branch should be taken when the loop is first run again. In summary, a 1-bit branch predictor mispredicts the first and last branches of a loop.

A 2-bit dynamic branch predictor solves this problem by having four states: *strongly taken*, *weakly taken*, *weakly not taken*, and *strongly not taken*, as shown in Figure 7.66. When the loop is repeating, it enters the “strongly not taken” state and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the “weakly not taken” state. When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and reenters the “strongly not taken” state. In summary, a 2-bit branch predictor mispredicts only the last branch of a loop.

As one can imagine, branch predictors may be used to track even more history of the program to increase the accuracy of predictions. Good branch predictors achieve better than 90% accuracy on typical programs.

The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer. By keeping track of both branch and jump destinations in the branch target buffer, the processor can also avoid flushing the pipeline during jump instructions.

7.8.3 Superscalar Processor

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.67 shows a block diagram of a two-way superscalar processor that fetches and executes two instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.

A *scalar* processor acts on one piece of data at a time. A *vector* processor acts on several pieces of data with a single instruction. A *superscalar* processor issues several instructions at a time, each of which operates on one piece of data.

Our MIPS pipelined processor is a scalar processor. Vector processors were popular for supercomputers in the 1980s and 1990s because they efficiently handled the long vectors of data common in scientific computations. Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors.

However, modern processors also include hardware to handle short vectors of data that are common in multimedia and graphics applications. These are called *single instruction multiple data (SIMD)* units.

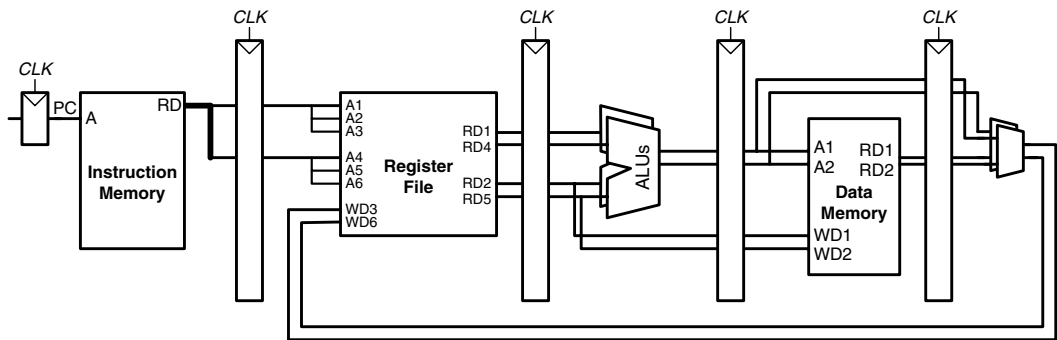


Figure 7.67 Superscalar datapath

Figure 7.68 shows a pipeline diagram illustrating the two-way superscalar processor executing two instructions on each cycle. For this program, the processor has a CPI of 0.5. Designers commonly refer to the reciprocal of the CPI as the *instructions per cycle*, or *IPC*. This processor has an IPC of 2 on this program.

Executing many instructions simultaneously is difficult because of dependencies. For example, Figure 7.69 shows a pipeline diagram running a program with data dependencies. The dependencies in the code are shown in blue. The `add` instruction is dependent on `$t0`, which is produced by the `lw` instruction, so it cannot be issued at the same time as `lw`. Indeed, the `add` instruction stalls for yet another cycle so that `lw` can forward `$t0` to `add` in cycle 5. The other dependencies (between

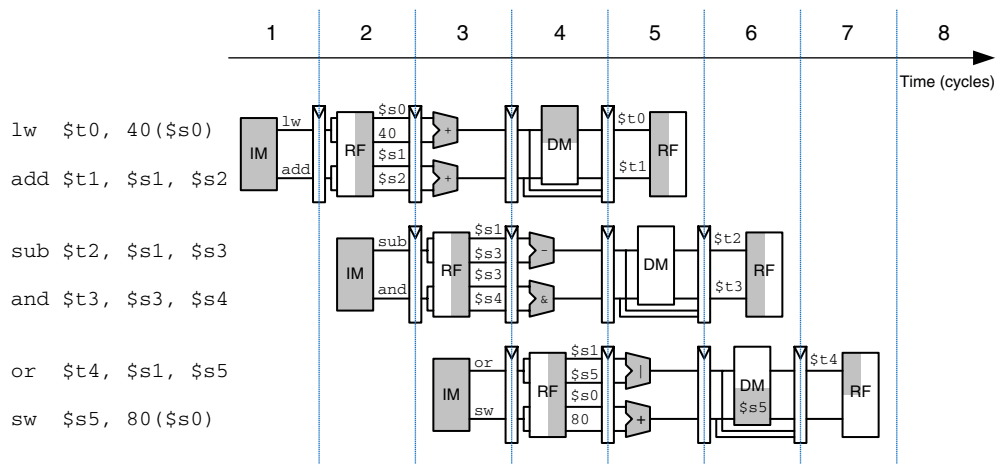


Figure 7.68 Abstract view of a superscalar pipeline in operation

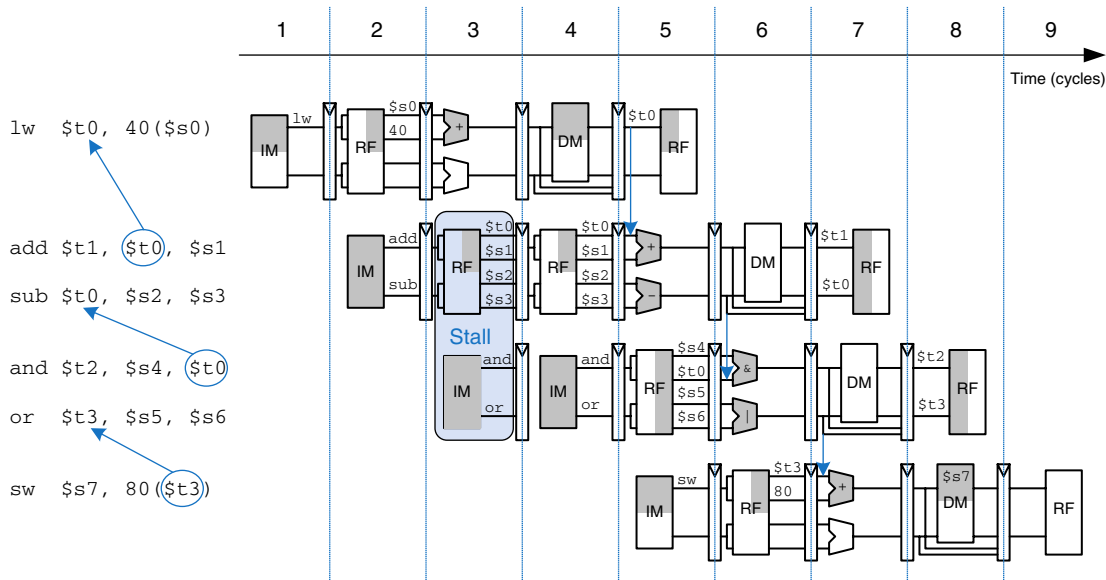


Figure 7.69 Program with data dependencies

sub and and based on \$t0, and between or and sw based on \$t3) are handled by forwarding results produced in one cycle to be consumed in the next. This program, also given below, requires five cycles to issue six instructions, for an IPC of 1.17.

```
lw $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. Multiple execution units is a case of spatial parallelism. Superscalar processors exploit both forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.

Commercial processors may be three-, four-, or even six-way superscalar. They must handle control hazards such as branches as well as data hazards. Unfortunately, real programs have many dependencies, so wide superscalar processors rarely fully utilize all of the execution units. Moreover, the large number of execution units and complex forwarding networks consume vast amounts of circuitry and power.

7.8.4 Out-of-Order Processor

To cope with the problem of dependencies, an *out-of-order processor* looks ahead across many instructions to *issue*, or begin executing, independent instructions as rapidly as possible. The instructions can be issued in a different order than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.

Consider running the same program from Figure 7.69 on a two-way superscalar out-of-order processor. The processor can issue up to two instructions per cycle from anywhere in the program, as long as dependencies are observed. Figure 7.70 shows the data dependencies and the operation of the processor. The classifications of dependencies as RAW and WAR will be discussed shortly. The constraints on issuing instructions are described below.

► Cycle 1

- The `lw` instruction issues.
- The `add`, `sub`, and `and` instructions are dependent on `lw` by way of `$t0`, so they cannot issue yet. However, the `or` instruction is independent, so it also issues.

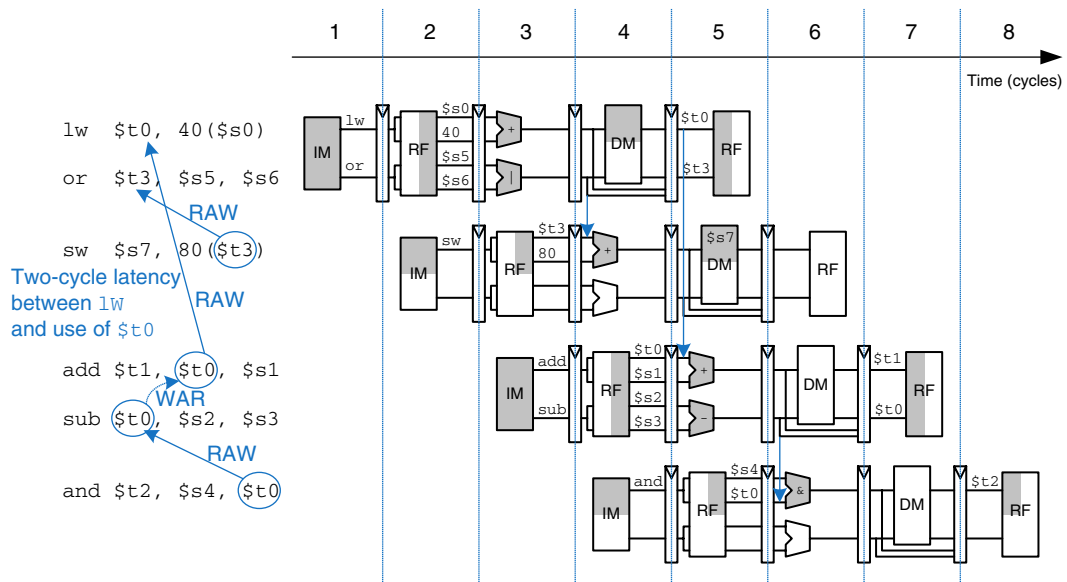


Figure 7.70 Out-of-order execution of a program with dependencies

- ▶ Cycle 2
 - Remember that there is a two-cycle latency between when a `lw` instruction issues and when a dependent instruction can use its result, so `add` cannot issue yet because of the `$t0` dependence. `sub` writes `$t0`, so it cannot issue before `add`, lest `add` receive the wrong value of `$t0`. `and` is dependent on `sub`.
 - Only the `sw` instruction issues.
- ▶ Cycle 3
 - On cycle 3, `$t0` is available, so `add` issues. `sub` issues simultaneously, because it will not write `$t0` until after `add` consumes `$t0`.
- ▶ Cycle 4
 - The `and` instruction issues. `$t0` is forwarded from `sub` to `and`.

The out-of-order processor issues the six instructions in four cycles, for an IPC of 1.5.

The dependence of `add` on `lw` by way of `$t0` is a read after write (RAW) hazard. `add` must not read `$t0` until after `lw` has written it. This is the type of dependency we are accustomed to handling in the pipelined processor. It inherently limits the speed at which the program can run, even if infinitely many execution units are available. Similarly, the dependence of `sw` on `or` by way of `$t3` and of `and` on `sub` by way of `$t0` are RAW dependencies.

The dependence between `sub` and `add` by way of `$t0` is called a *write after read* (WAR) hazard or an *antidependence*. `sub` must not write `$t0` before `add` reads `$t0`, so that `add` receives the correct value according to the original order of the program. WAR hazards could not occur in the simple MIPS pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, `sub`) is moved too early.

A WAR hazard is not essential to the operation of the program. It is merely an artifact of the programmer's choice to use the same register for two unrelated instructions. If the `sub` instruction had written `$t4` instead of `$t0`, the dependency would disappear and `sub` could be issued before `add`. The MIPS architecture only has 32 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all the other registers are in use.

A third type of hazard, not shown in the program, is called *write after write* (WAW) or an *output dependence*. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it. The hazard would result in the wrong value being

written to the register. For example, in the following program, `add` and `sub` both write `$t0`. The final value in `$t0` should come from `sub` according to the order of the program. If an out-of-order processor attempted to execute `sub` first, the WAW hazard would occur.

```
add $t0, $s1, $s2
sub $t0, $s3, $s4
```

WAW hazards are not essential either; again, they are artifacts caused by the programmer's using the same register for two unrelated instructions. If the `sub` instruction were issued first, the program could eliminate the WAW hazard by discarding the result of the `add` instead of writing it to `$t0`. This is called *squashing* the `add`.⁶

Out-of-order processors use a table to keep track of instructions waiting to issue. The table, sometimes called a *scoreboard*, contains information about the dependencies. The size of the table determines how many instructions can be considered for issue. On each cycle, the processor examines the table and issues as many instructions as it can, limited by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.

The *instruction level parallelism (ILP)* is the number of instructions that can be executed simultaneously for a particular program and microarchitecture. Theoretical studies have shown that the ILP can be quite large for out-of-order microarchitectures with perfect branch predictors and enormous numbers of execution units. However, practical processors seldom achieve an ILP greater than 2 or 3, even with six-way superscalar datapaths with out-of-order execution.

7.8.5 Register Renaming

Out-of-order processors use a technique called *register renaming* to eliminate WAR hazards. Register renaming adds some nonarchitectural *renaming registers* to the processor. For example, a MIPS processor might add 20 renaming registers, called `$r0-$r19`. The programmer cannot use these registers directly, because they are not part of the architecture. However, the processor is free to use them to eliminate hazards.

For example, in the previous section, a WAR hazard occurred between the `sub` and `add` instructions based on reusing `$t0`. The out-of-order processor could *rename* `$t0` to `$r0` for the `sub` instruction. Then

⁶ You might wonder why the `add` needs to be issued at all. The reason is that out-of-order processors must guarantee that all of the same exceptions occur that would have occurred if the program had been executed in its original order. The `add` potentially may produce an overflow exception, so it must be issued to check for the exception, even though the result can be discarded.

sub could be executed sooner, because \$r0 has no dependency on the add instruction. The processor keeps a table of which registers were renamed so that it can consistently rename registers in subsequent dependent instructions. In this example, \$t0 must also be renamed to \$r0 in the and instruction, because it refers to the result of sub.

Figure 7.71 shows the same program from Figure 7.70 executing on an out-of-order processor with register renaming. \$t0 is renamed to \$r0 in sub and and to eliminate the WAR hazard. The constraints on issuing instructions are described below.

- ▶ **Cycle 1**
 - The lw instruction issues.
 - The add instruction is dependent on lw by way of \$t0, so it cannot issue yet. However, the sub instruction is independent now that its destination has been renamed to \$r0, so sub also issues.
- ▶ **Cycle 2**
 - Remember that there is a two-cycle latency between when a lw issues and when a dependent instruction can use its result, so add cannot issue yet because of the \$t0 dependence.
 - The and instruction is dependent on sub, so it can issue. \$r0 is forwarded from sub to and.
 - The or instruction is independent, so it also issues.

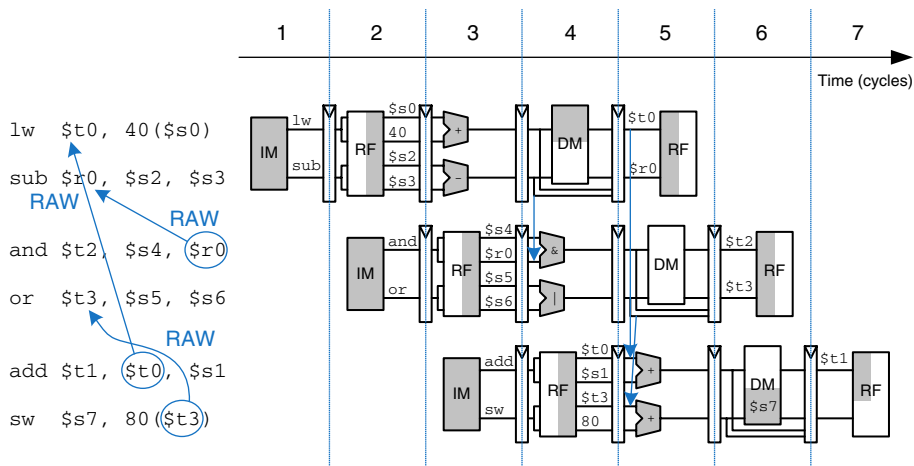


Figure 7.71 Out-of-order execution of a program using register renaming

► Cycle 3

- On cycle 3, `$t0` is available, so `add` issues. `$t3` is also available, so `sw` issues.

The out-of-order processor with register renaming issues the six instructions in three cycles, for an IPC of 2.

7.8.6 Single Instruction Multiple Data

The term *SIMD* (pronounced “sim-dee”) stands for *single instruction multiple data*, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called *packed* arithmetic.

For example, a 32-bit microprocessor might pack four 8-bit data elements into one 32-bit word. Packed add and subtract instructions operate on all four data elements within the word in parallel. Figure 7.72 shows a packed 8-bit addition summing four pairs of 8-bit numbers to produce four results. The word could also be divided into two 16-bit elements. Performing packed arithmetic requires modifying the ALU to eliminate carries between the smaller data elements. For example, a carry out of $a_0 + b_0$ should not affect the result of $a_1 + b_1$.

Short data elements often appear in graphics processing. For example, a pixel in a digital photo may use 8 bits to store each of the red, green, and blue color components. Using an entire 32-bit word to process one of these components wastes the upper 24 bits. When the components from four adjacent pixels are packed into a 32-bit word, the processing can be performed four times faster.

SIMD instructions are even more helpful for 64-bit architectures, which can pack eight 8-bit elements, four 16-bit elements, or two 32-bit elements into a single 64-bit word. SIMD instructions are also used for floating-point computations; for example, four 32-bit single-precision floating-point values can be packed into a single 128-bit word.

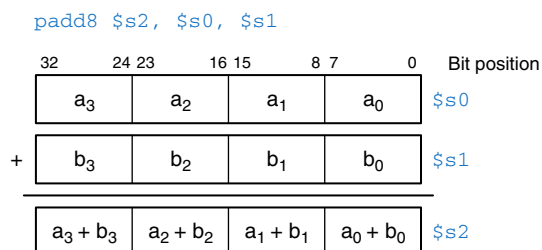


Figure 7.72 Packed arithmetic: four simultaneous 8-bit additions