_____

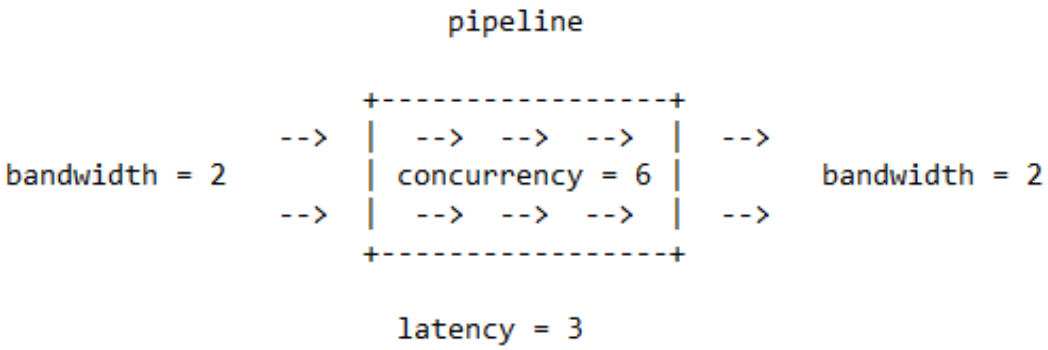
RISC Instruction Execution

_____

Little's law

_____

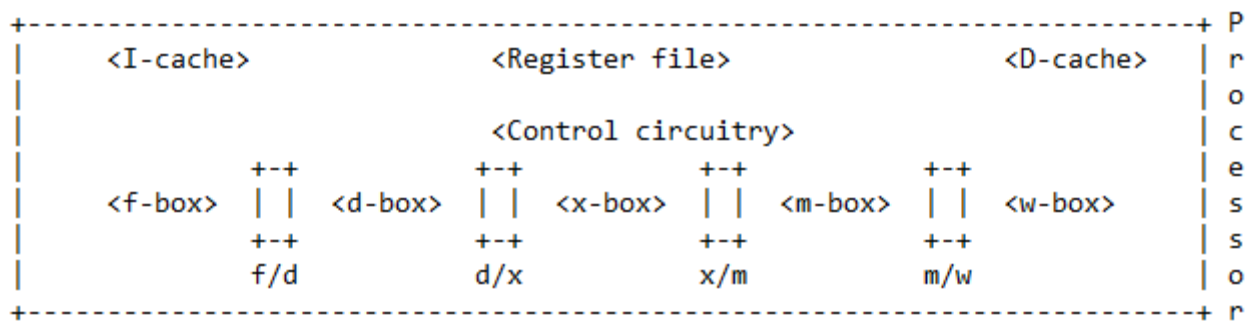
Pipelines take time to reach their equilibrium state. We may feed operations into an empty pipeline but need to wait until we get our first result. It is only after the pipeline has reached its equilibrium state that the result bandwidth on the right will match the input bandwidth on the left. Here is a picture:



Suppose initially the pipeline is empty and we apply a sustained input bandwidth of two operations per cycle. Eventually, the pipeline will reach equilibrium, and we will receive a sustained output of two results per cycle. In the picture, it is not specified whether we are supplying peak input bandwidth. In general (Little's Law), at equilibrium, the occupancy (concurrency) of the pipeline is the latency-bandwidth product.

MIPS RISC 1.0 pipeline

_____



Boxes are shown using angular brackets and box names. Latches are shown as short rectangles. Crudely, you may view boxes as combinational logic, and latches as collections of pipeline registers. Job descriptions follow.

1. The f-box reads the memory address of the next instruction from the PC register, and fetches it from memory, or from the I-cache. It then updates PC by adding 4. Each box must complete its work, including all latching, in one clock cycle. Here, the fetched instruction is latched in the f/d latch.

2. The d-box has multiple tasks. It must decode the instruction, noting the operands (register and immediate), and the destination register (if any). It also localizes any register operands in the instruction from the register file. The d-box also processes conditional branches. Consider 'bne r1,r2,loop'. The d-box tests inequality of the two registers it has localized. If they are unequal, the branch is deemed taken, and the d-box adds the offset (a multiple of the immediate 'loop') to the base register PC. A number of things are latched by the d-box in the d/x latch.

3. The x-box is actually a case statement, which requires control input.

i) In a memory reference, we add the base register and the offset to compute the memory address.

ii) In a register-register instruction, we perform the operation specified by the opcode.

iii) In a register-immediate instruction, we perform the operation specified by the opcode, using the immediate as one operand value.

Think of the x-box as a collection of functional units, perhaps an integer unit, a floating-point unit, a register shifter, a logic unit, etc. So the x-box begins to seem somewhat ghostly if you think about it too carefully. The x-box result is latched in the x/m latch.

4. The m-box is another case statement and requires control. If the instruction is a load, the m-box reads from memory, or from the D-cache, and latches the result in the m/w latch. If the instruction is a store, the m-box writes to memory, or to the D-cache, taking the value to be stored from some pipeline latch. Otherwise, the m-box does nothing. When data flows from the x-box to the w-box during such a case, this is called "bypassing".

5. The w-box does the same thing if the instruction is a load, or an ALU instruction, which necessarily produces a result; namely, it takes either the loaded value, or the ALU result, either of which can be found in the m/w latch, and writes it to the destination register.

Not all instructions use all five boxes. A conditional branch uses only the f-box and the d-box. A store uses f, d, x, and m. An ALU instruction uses f, d, x, and w. Only a load uses f, d, x, m, and w.

Boxes and Latches Form a Pipeline (in all modern computers)

_____

```
         1 2 3 4 5 6 7 8 9
instr 1  f d x m w
instr 2    f d x m w
instr 3      f d x m w
instr 4        f d x m w
instr 5          f d x m w
...                ...
```

There is one unusual point. Since there are cycles in which both the d-box and the w-box access the RF, we _write_ in the first half of the cycle, and _read_ in the second half of the cycle.

Latches are also an intrapipeline _communication mechanism_. Any value latched by the end of cycle 'n' moves to the next latch by the end of cycle 'n+1'.

This is called _fowarding_. Think of the pipeline as a river which is constantly flowing to the right.

Sometimes the pipeline must _stall_. There are three possible reasons:

a) hardware unavailable ("structural hazard")

b) data unavailable ("data-dependance hazard")

c) branch decision unavailable ("control hazard")

How bad are stalls? Answer: $su = \frac{d}{1 + s/i}$. We need to minimize stalls.

More precisely, the speedup due to pipelining is the depth of the pipeline divided by 1 plus the average number of stall cycles per instruction.

In RISC 1.0, the only data dependence that can cause problems is the _flow dependence_, in which an earlier instruction produces a value that is consumed by a later instruction. (In RISC 2.0, we need to worry about the other two data dependences). Actually, the MIPS RISC 1.0 pipeline with multicycle operations can have other types of data dependence, but we will just ignore this.

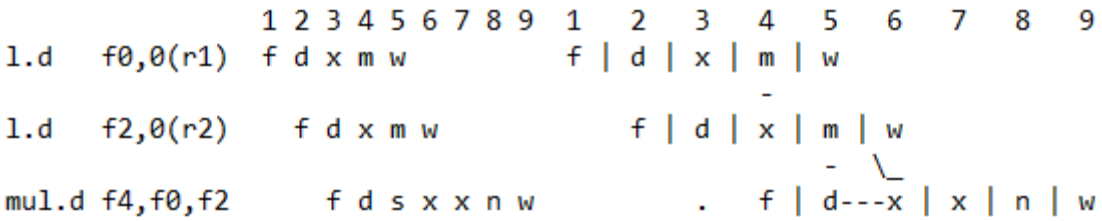
Sometimes we get lucky. Consider:

```
                1 2 3 4 5 6 7 8 9  1   2   3   4   5  6  7  8  9
add r1,r2,r3    f d x n w          f | d | x | n | w
add r4,r1,r4      f d x n w            f   d  \x  \n  w
add r5,r1,r5        f d x n w            f   d   x  n  w
add r6,r1,r6          f d x n w              f   d--x  n  w
add r7,r1,r7            f d x n w                f  d--x  n  w
```

Let's follow 'r1'. It is in the x/m latch at the start of cycle 4. It is in the m/w latch at the start of cycle 5. It is sent to the RF in cycle 5, but also retrieved from the RF in cycle 5, and available in the d/x latch at the start of cycle 6. Finally, 'r1' is retrieved from the RF in cycle 6 and is available in the d/x latch at the start of cycle 7.

There are no stalls here because the x-box is in the middle.

At other times we have no choice but to stall. Because I hate oversimplification, I am going to give 'mul.d' two x-boxes (for now)..
This is more interesting. 'f0' is sent to the RF in cycle 5, but also retrieved from the RF in cycle 5, and available in the

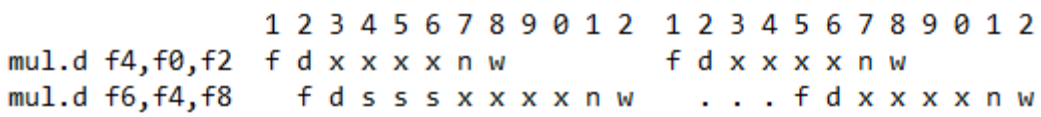


d/x latch at the start of cycle 6. 'f2' is in latch m/w at the start of cycle 6, and is retrieved from that latch in cycle 6.

There is a _load stall_ here because the m-box is slightly to the right.

Does this seem like black magic? The rule is: If a box will perform an operation in cycle 'n', and one of its operands is available in a latch at the start of cycle 'n', then the box may retrieve and use that operand in cycle 'n'. A box may retrieve two operands from the same or different latches in the same cycle.

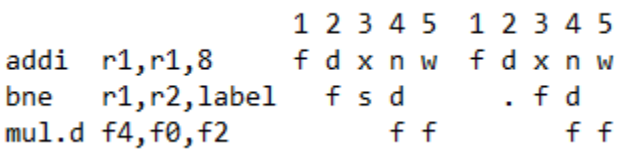
Here is another example. Now, I have put 4 x-boxes in 'mul.d'.



This is a simple latch retrieval. In cycle 7, the x-box pulls 'f4' out of the x/m latch which is available there at the start of cycle 7.

All the dependences we have seen so far are _flow dependences_, also known as _producer-consumer dependences_.

Here is an example of a branch stall, together with a flow-dependence stall.



Here, I have drawn the case where the branch is taken. The 'mul.d' is fetched in cycle 4, but this is not the next instruction that will be executed. So we have to fetch again to get the instruction with the label 'label'. In the real world, branches are sometimes taken and sometimes not taken. In my notes, I always draw the worst-case scenario where a second fetch is required. This is called a _branch stall_.

There is a branch stall because the d-box is slightly to the left.

The policy illustrated here is the _predict-not-taken_ policy. The processor fetches the next instruction as if the branch is not taken. If that turns out to be true, there is no branch stall. Only if the branch is taken is it necessary to refetch.

<* material not covered last time *>

It is time to be more honest about the x-box. Since it is really a virtual collection of different functional units, we may have two x-boxes active in the same cycle. Consider:

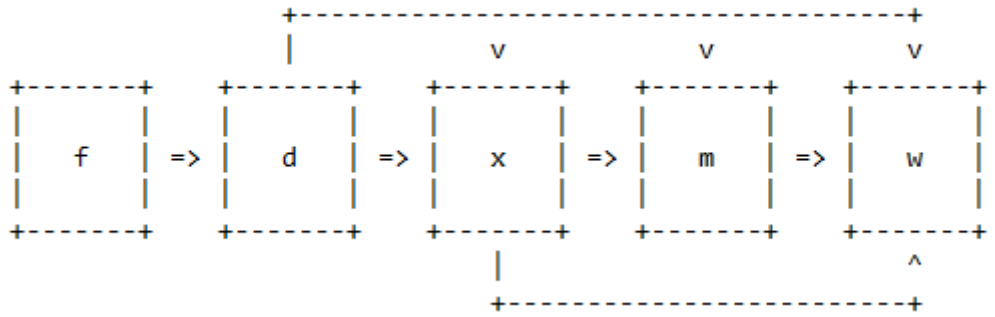
```
                  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3
l.d   f0,0(r1)    f | d | xi| m1| m2| w |   |   |   |   |   |   |
                  ------------------------------------------------
l.d   f2,0(r2)      | f | d | xi| m1| m2| w |   |   |   |   |   |
                  ------------------------------------------------
mul.d f4,f0,f2      |   | . | . | f | d | x1| x2| x3| x4| n | w |
                  ------------------------------------------------
s.d   f4,0(r1)      |   |   |   |   | . | . | f | d | xi| m1| m2|
```

Here, memory references have two m-boxes, and floating-point multiplies have four x-boxes. What this means is that I have replaced two pipeline boxes by embedded _internal pipelines_. The numbers are self-explanatory. Remember that address calculation is an integer operation, which can be handled by the integer-arithmetic functional unit, here called 'xi'. In cycle 10, there is no "structural" conflict between 'xi' and the floating-point-arithmetic functional unit, which is still active.

We make the assumption that all arithmetic functional units, whether one box or many, are _fully pipelined_. That is, you may feed in a new arithmetic task to the unit in each and every cycle.

Moreover, there is no "structural" conflict between a box doing nothing, and that same box doing something. This situation sometimes occurs with the m-box.

Nonpipelined RISC 1.0 Datapath (use this model for Assignment 3)



We have been focusing on pipelining. However, Assignment 3 asks you to emulate the _nonpipelined_ RISC 1.0 datapath. Here, I draw some previous material together to help you see the big picture. Clearly, in any emulation, we are going to idealize certain things that matter for real computers, but whose details have no pedagogical value. The first thing we idealize is pretending that register designators and immediate are encoded using the same number of bits (in instruction formats), which simplifies programming, but makes no conceptual difference.

With equal sizes, the only thing that matters is the left-to-right ordering.

1. Instruction Formats (opc, arg1, arg2, arg3) // opcode and three arguments

```
l.d   f2,4(r1)  encodes as:  opc, r1, f2,  4
l.d   f4,8(r3)  encodes as:  opc, r3, f4,  8
mul.d f6,f2,f4  encodes as:  opc, f2, f4, f6
add.d f8,f4,f6  encodes as:  opc, f4, f6, f8
s.d   f6,8(r3)  encodes as:  opc, r3, f6,  8
s.d   f8,4(r5)  encodes as:  opc, r5, f8,  4
bne   r1,r3,lab encodes as:  opc, r1, r3,  0  // fake offset (no goto)
bne   r3,r3,lab encodes as:  opc, r3, r3,  0  // fake offset (no goto)
```

I use decimal digits to encode arguments to not be distracted by string processing. They are the digits you see above, but I have added 'r' and 'f' in front of some of them for human convenience. In the program, all 'odd' registers are r-registers, and all 'even' registers are f-registers, so no lower-case Roman letters appear in the _argument_ object code. (I put some of these letters in my print statements, again, for human convenience). In this scheme, both immediate and register designators are single decimal digits.

The object code would then be (if this were the program):

```
1124     (output as: l|124)
1348     (output as: l|348)
m246     (output as: m|246)
a468     (output as: a|468)
s368     (output as: s|368)
s584     (output as: s|584)
b130     (output as: b|130)
b330     (output as: b|330)
```

2. Box Behavior

f-box: Kind of obvious. By the way, my machine has "instruction-addressed" memory (all integers are in units of instructions!), so the 'for' loop has the correct increment.

d-box: Decode (decompose) instruction. Set 'opc'. Localize all register operands from the RF. Set either 'dreg' (destination register) or 'sval' (the value to store). Set the two data outputs from the d-box to the x-box ('D_Out1' and 'D_Out2'). This is a case statement because you need to know the value of 'opc' in order to choose the right behavior. There are no extra marks for the elegance of your case statements.

The d-box also handles conditional branches. Since your languages don't have gotos, an untaken branch is a no-op, while a taken branch causes you to exit the 'for' loop. That's why I encoded the offset as 0. Set 'branch' to the branch decision.

x-box: Calculate the memory address or perform the specified arithmetic instruction. Set 'X_Out' (the data output from the x-box). Remember: There are only integers in the emulation, even if they sometimes stand for floating-point numbers in the real world. Interrogate 'opc'.

m-box: Either bring in the floating-point value from memory at the memory address or use a previously localized operand to push a value to memory at the memory address. Set 'M_Out' (the data output from the m-box, if there is one). Interrogate 'opc'.

w-box: If there is a destination register (interrogate 'opc'), push the appropriate result value to it, perhaps a loaded value, perhaps a computed value. In fact, such a value could have come from either of the two boxes.

Please note that both the m-box and the w-box may do nothing during a cycle, depending on 'opc'. I added the print statement "Did nothing." to several of my boxes as a default option, because it gives a clearer big picture. I suggest that you do the same.

You understand that your program's output is an essential window into your source code. So, I insist on a single, integrated plaintext file consisting of your source code, to which your output has been appended as a comment. In that way, we can run your program and see if your program _does_ produce the output it purports to output. By the way, a Word document doesn't compile.

Compilers are like me: they prefer plaintext. :-)

Submit your assignment to Moodle ("Programming Assignment 3").

Caution: My descriptions of box actions describe the results of your case statements, but are not necessarily isomorphic to them.

3. Walkthrough of the Assignment-Provided d-box Code

Warning: The invariants you use to design your code hold for _nonpipelined_ datapaths. They may not hold for RISC 1.0 pipelines, such as the one on the midterm and the one on the final.

Look at the instruction formats. In all cases ('l','s','a','m','b'), 'arg1' is the specifier of an operand register that needs to be localized. For this reason, in a previous version, the d-box code _had_ the option of localizing this register value, and sending that value to a d-box output port before entering the case statement---when there were no conditional branches! It is more accurate in the current program to include this action in every appropriate case.

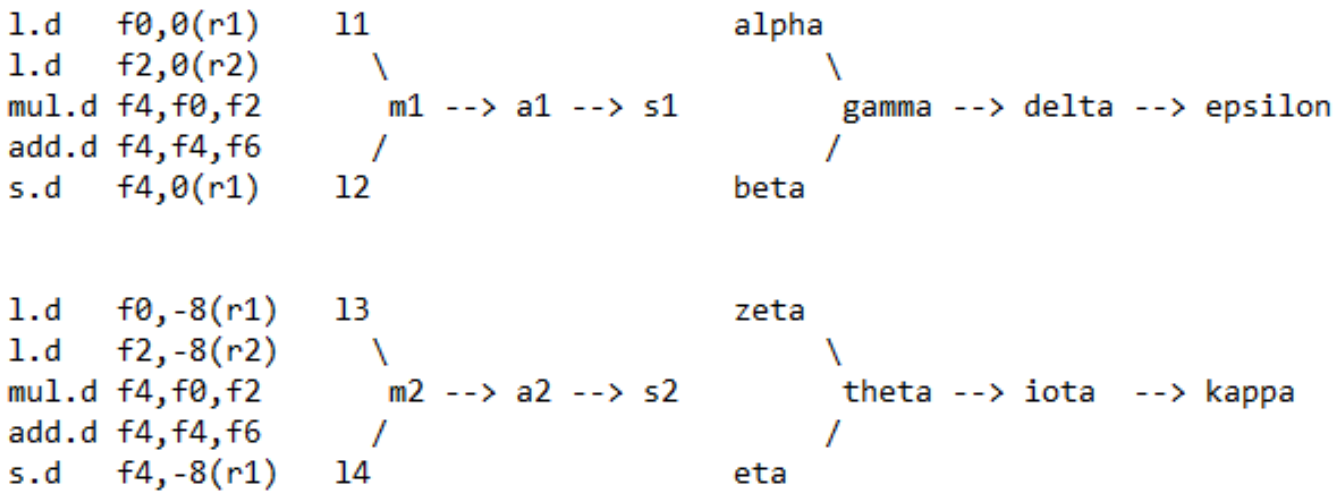
Now, suppose the case is an arithmetic statement. In this case, 'arg2' is also the specifier of an operand register that needs to be localized. Hence, we perform the same action as above. But now, in addition, we need to set 'dreg' to the specifier of the destination register, in this case, 'arg3'.

This is all the code I provided. You finish the d-box code by deciding what to do in the case of a load, what to do in the case of a store, and what to do in the case of a conditional branch. For the remaining boxes, you will do something similar, but will now take advantage of the global variables (all shown in the assignment) that have been given a value by earlier boxes.

You need to finish Assignment 2 first, but it is not a bad idea to take a look at Assignment 3 long before it is due.

Brief look at RISC 2.0

Consider the first two iterations of a loop, where only the loop bodies are shown. I have adjusted the memory addresses that will be stored in the load buffer and the store buffer, assuming the loop is proceeding leftward through each of two arrays.



Think of the nodes labeled with Greek letters as "Zoom waiting rooms". Now, the data dependences in the program mean that some instructions may not issue until their flow-graph predecessors have completed. Imagine all ten instructions in waiting rooms. All four loads are ready to issue now. But suppose at least one of the first two loads is slow, while both second two loads are fast. Then the multiply in the second iteration can issue.

Notice that this could not happen if we executed the program in (von Neumann) program order. Instead, we are stuck waiting for the slow load (or loads) to complete. It is this extra freedom that is the principle change when we move from RISC 1.0 to RISC 2.0.

It sounds great, but there are some serious downsides.

Information Flow

Consider the x-, m-, and w-boxes. Here is a brief description of the data flow. If the x-box is to act on some value or values, it must receive it (or them). Also, it may receive a further data value that specifies more precisely the action to be performed (e.g., the number of bits to shift).

In a store instruction, the m-box receives a data value from the pipeline, namely, the value to be stored. It also receives a data value that is the memory address of where in memory the first value is to be stored. In a load instruction, the m-box only receives the memory address. When the instruction contains a destination register, the w-box receives both a data value and a register name from the pipeline. This is what is to be written back and where it is to be written back to. Obviously, one only feeds data to a box when one intends that box to do something.

Reality Check

_____

Consider the f-box. Allegedly, it sends PC to memory and fetches the next instruction. Although we don't call this a _memory reference_, i.e., a load or a store, it certainly is a _memory access_. How many processor cycles does it take to access memory on a typical computer? In lecture 1, I suggested the figure of 200 cycles. You cannot address memory and get back a word in a single cycle. What is going on here?

In reality, the f-box sends PC to the _instruction cache, or _I-cache_. A cache is a complex state element that can store copies of some, but obviously not all, memory locations. Most caches (L1$, L2$, L3$) are small enough to fit on the processor chip. So, we have to pretend two things that are not true. First, that it is possible to access the I-cache in a single cycle. Second, that the I-cache magically _always_ has a copy of just the instruction we need. Surprisingly, for reasons explained later, these two white lies do not grossly misrepresent the actual performance of the I-cache.

Hint: this good fortune has something to do with the fetch-execute cycle.

Consider the m-box. Allegedly, it either sends a data address to memory and receives a data value (this is a load), or else it sends a data address plus a data value to memory, thus depositing the data value (this is a store).

Recall that the m-box only does work when the pipeline is executing a memory reference. Still, each memory reference is certainly a memory access. Do we have a problem here?

In reality, the m-box interacts with the _data cache_, or _D-cache_. This is another complex state element that can store copies of some, but obviously not all, memory locations. Again, we pretend two things that are not true.

- First, that it is possible to access the D-cache in a single cycle.
- Second, that the D-cache magically _always_ has a copy of the data value we wish to load or store. For data, these two white lies can cause _serious_ problems. It depends on the memory-accessing pattern.

Consider program P executing during some interval I. Suppose that, during this interval, program P only uses data from, say, 20K distinct memory locations. We say that these 20K locations constitute the _working set_ of program P during interval I. If program P's working set fits into the computer's D-cache, then we can't really say we have a problem. However, if the working set doesn't fit into the D-cache, then we may well have a problem: we may repeatedly fail to find the data copy we want in the D-cache, and thus may repeatedly be forced to access the actual memory, at much greater cost. Which of these two cases is the real one depends on the memory-accessing pattern of program P. Some patterns generate enormous working sets that are too big to fit into any D-cache.

This problem even has a name: it is officially called the _Memory Wall_.

Consider Moore's Law. It says that processor performance, which we may roughly model as the product of the number of logic transistors times the clock frequency in Hz, increases exponentially over time. Before 2003, both factors increased exponentially. Now, we have to be very careful with the clock frequency. Still, replacing a single power-inefficient core with many power-efficient cores increases the power efficiency of the processor chip as a whole, and thereby allows us to increase performance without overstepping our power budget. At least, this was the original multicore dream. Initially, multicore was a mess because no one know how to build network-on-a-chip (NoC) chip interconnection networks. Or how to handle cache coherence. One question everyone can ask is, in the next generation, do I get twice as many cores, or only two more cores? I don't know anyone who thinks that multicore, by itself, can be a driver of exponential growth in processor performance.

Increased arithmetic performance is only possible if there is increased delivery of data operands to functional units. Memories are making some improvements to memory latency and memory bandwidth, but not fast enough to keep pace with processor improvements. Raw processor performance increases faster than raw memory performance. An ideal, or perfect, cache could easily compensate for the mismatch between processor demand and memory supply.

However, in the real world, caches are _not_ ideal, and programs are not always _cache friendly_. In the worst case, program performance may be limited by memory performance, and be unaffected by increases in processor performance. All moderately educated computer professionals agree that the Memory Wall and the Power Wall are the two main challenges that must be overcome by today's computer designers. Dealing with the Power Wall is mandatory. What about the Memory Wall? We could accept to only write programs that play nicely with today's caches. But this concession might be selling our birthright for a mess of pottage.

Review of Pipelining

Pipelining is a naturally efficient way for a datapath to execute machine instructions. It was standard in vector supercomputers. The basic idea is to allow different workstations ("boxes") to work on different instructions at the same time. By overlapping the execution of different machine instructions, we can make significant improvements in the pipeline's execution _throughput_.

Say that executing some machine instruction is a task 'T'. Suppose we break 'T' into a _sequence_ of nonoverlapping subtasks: 'T = t1; t2; ...; tn'. Now, we provide a specialized workstation for each subtask (these are our boxes). Think of the processor clock as ringing a bell at the start of each processor cycle. For simplicity, let's stick with having five stages, with the f, d, x, m, and w boxes as our specialized workstations. An instruction pipeline doesn't have to have five stages---it could have 21---but this was the design of an early RISC processor.

When the bell rings, each workstation passes the partially executed instruction to the workstation on its right.

Is this advantageous? Suppose each box completes its work in one clock cycle. Therefore, the time to execute an instruction (the instruction _latency_) is 5 cycles. But when the pipeline achieves cruising speed, a new instruction completes every cycle, giving the pipeline an execution _bandwidth_ of one instruction per cycle. (We call this _single-cycle pipelining_). The ideal speedup when a pipeline is pipelined is thus equal to the pipeline _depth_ (the number of stages).

Each of our five boxes is a _combinational_ circuit, but the clocked pipeline as a whole is a _sequential_ circuit. To make this work, each box is followed immediately on its right by a set of (nonISA) _pipeline registers_, which is called a "pipeline latch". The basic requirement is this: Prior to the end of a clock cycle, all the results from a given stage must be stored in the pipeline latch to its right, in order that these values can be preserved across clock cycles, and used as inputs to the next stage at the start of the next clock cycle.

Note: In this course, we may _pretend_ that pipeline boxes are combinational logic, but in the real world they are not _pure_ combinational logic.

Also, it is reasonable to think of the boxes as combinational circuits that compute Boolean functions, and of the latches as complementary state elements that allow the pipeline to function as a finite state machine.

Finally, there is a notion of _gap_, which is the number of stalls between two adjacent flow-dependent instructions; obviously, it depends on the identity of both the producer and the consumer instruction.

Here are some examples.

```
a) l.d f4,...  f d x m w
   s.d f4,...    f d x m          gap = 0

   l.d   f4,...    f d x m w
   mul.d f6,f4,...   . f d x ...  gap = 1
```

```
b) mul.d f6,...  f d x x x x n w
   s.d   f6,...    . . f d x m       gap = 2

   mul.d f6,...    f d x x x x n w
   mul.d f8,f6,...   . . . f d x ...  gap = 3
```