Lecture 5 Computer Organization and Design DKP/Fall 2020

\_\_\_\_\_\_\_\_\_

It is not immediately obvious that Moore's law should favor one type of

processor architecture over another. Why, starting in 1980, did killer

micros have such success? Which processor architectures died off? What

is a killer micro, anyway? Does \_every\_ microprocessor count as a killer

micro? Here is a subjective answer.

The Cray vector architectures died out because they were too expensive,

and also because parallel vector processors (i.e., computers) ran into

trouble scaling to larger number of processors. But parallel vector

processors were supercomputers, pioneering advanced architectural ideas

with exotic technologies. More down to earth, DEC's Vax architecture ran

into trouble because its complex instruction set made it very hard to

implement fast pipelining. (The Vax can stand for other machines that

lost their appeal). In 1980, the idea of RISC (reduced instruction set

computing) architecture gained ground, and led to radical simplifications

in the implementation of pipelining.

In the 1970s, machines had hundreds of different instructions, in a variety

of formats, leading to thousands of distinct combinations when

addressing-mode variations were taken into account. Interpretation of all

possible combinations of opcodes and operands required very complex control

circuitry. Early VLSI chips simply did not have enough room to hold both

this massive control circuitry and a register file with sufficiently many

registers. RISC machines were successful because they radically reduced

the amount of control circuitry, i.e., they reduced the control \_overhead\_.

Moreover, the simplicity of control in RISC machines allowed a fully

hardwired implementation of the control circuitry, breaking with the

tradition of microprogrammed control. This dramatically increased

execution speed.

The distinction between RISC and CISC has not stayed hard and fast since

1980, especially after the introduction of Intel chips in the 1990s that

were CISC on the outside, and RISC on the inside. In all fairness, we have

to count more recent Intel and AMD processors as killer micros. This is so

even if they have processor architectures only a mother could love. Another

concern is that CISC processors are incredibly expensive and time-consuming

to design. You almost have to admire Intel for not being killed by the

complexity of its own designs. Whether this is a good starting point for

multicore, or, indeed, for any \_system on a chip\_, is another question

entirely. Finally, describing CISC architectures to students is not exactly

enjoyable.

We can sketch some of the core ideas in RISC design philosophy, which is

both an approach to instruction-set design, and a set of implementation

strategies.

1. There shall be a small set of instructions, each of which can be executed

in approximately the same amount of time using hardwired control (you may

need several RISC instructions to do the work of one complex instruction).

2. The architecture shall be a \_load/store\_ architecture that confines

memory-address calculation, and memory-latency delays, to a small set of

load and store instructions, with all other (register-register) instructions

obtaining their operands from faster, and compactly addressable, processor

registers.

3. There shall be a limited number of simple addressing modes that eliminate

or speed up address calculations for the vast majority of cases.

4. There shall be simple, uniform instruction formats that facilitate

extraction/decoding of the various fields. This allows overlap between

opcode interpretation and register readout.

Admittedly, what ultimately counts are the run times of our programs. What

matters is not whether the computer executes 10 billion simple instructions,

or 5 billion complex instructions, but simply what the total run time is.

We can characterize RISC architectures in slightly different language

(redundancy is not bad):

1. All operations on data apply to data in registers.

2. The only operators that affect memory are loads (which move data from

memory to a register) and stores (which move data from a register to memory).

3. The instruction formats are few in number, with all instructions typically

being one size.

RISC architectures include MIPS, ARM, PowerPC, PA-RISC, SPARC, and the

now-defunct Alpha. However, if MIPS and ARM own the embedded market, the

most powerful large-scale computers today appear to be hybrids of Intel chips,

or AMD chips, and Nvidia chips. Why this is so is a long story.

Terminology

\_\_\_\_\_\_\_\_\_\_\_

Early descriptions of processor microarchitecture emphasized the distinction

between (active) control circuitry and (passive) datapath circuitry that just

followed orders. As time went on, many control functions were incorporated

into the datapath. Even so, the conceptual need for distinct control circuitry

remained because, while we can imagine an intelligent datapath that can analyze

and control aspects of executing individual instructions, only separate control

circuitry can maintain a more global picture of how different instructions

should interact. I somewhat deemphasize the distinction between datapath and

control because i) I have no interest in drawing circuit diagrams showing

data and control wires, and ii) in 2020, it is nearly impossible to say where

the datapath ends and the control circuitry begins.

Moreover, all datapaths soon became pipelined. Today, it is more natural to

speak of an instruction-execution pipeline with some control functionality that

makes use of various on-chip hardware resources and is ultimately guided by

control circuitry that is \_notionally\_ outside the pipeline.

RISC pipelines were immediately successful yet they did not stand still.

Rather, they continued to evolve as heavier and heavier performance demands

were placed on them. This continued until the first RISC inflection point in

the 1990s. In fact, the RISC microachitecture dominant in the 21st century is

radically different from the one dominant in the 20th century (the multilevel

cache was the troublemaker). RISC 1.0 has in-order instruction-execution

pipelines: machine instructions are executed in \_program order\_. RISC 2.0

has out-of-order instruction-execution pipelines: machine instructions are

executed \_out of program order\_ using dataflow ideas to overcome the strict

von Neumann control flow in RISC 1.0. We start with the simpler RISC 1.0.

RISC Instruction Execution

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

We now study the MIPS RISC instruction-execution pipeline (at least in the

form that was popular in the 1980s). We waste no time on nonpipelined

datapaths. First,we need the concept of \_pipeline\_.

Consider a computer system that takes in operations on the left, computes

them, and then pushes out results on the right. In a pipeline, we may push

in new operations on the left long before getting the results of previous

operations pushed out on the right. A pipeline is characterized by three

parameters: First, there is the peak input bandwidth (the maximum rate the

pipeline will tolerate). Second, there is the operation latency (the time

for an operation to complete). Third, there is the pipeline occupancy (the

number of uncompleted operations in the pipeline at any one time).

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 equilbrium state that the result

bandwidth on the right will exactly match the input bandwidth on the left.

Here is a picture:

pipeline

+-----------------+

--> | --> --> --> | -->

bandwidth = 2 | concurrency = 6 | bandwidth = 2

--> | --> --> --> | -->

+-----------------+

latency = 3

Suppose initially the pipeline is empty and we apply a sustained input

bandwidth of two operations per cycle. Eventually, the pipeline will reach

equlilibrium and we will receive a a sustained output of two results per

cycle. In the picture, it looks like we are supplying peak input bandwidth

but I never said this. In general (Little's Law), at equilibrium, the

occupancy (concurrency) of the pipeline is the latency-bandwidth product.

Pipelining has performance consequences. In the picture above, system

throughput (result bandwidth) is 2 results per cycle. In contrast, if we

were to wait for the previous result(s) before supplying new input,

throughput would drop to 2/3 result per cycle. In general, pipelining a

system multiplies the throughput by the latency.

The MIPS 'fdxmw' instruction-execution pipeline is a special case of the

general pipeline sketched above. At equilibrium, the input bandwidth is 1,

the output bandwidth is 1, the latency is 5, and the occupancy (concurrency)

is 5.

Here is a picture of the MIPS RISC 1.0 pipeline.

+-------------------------------------------------------------------------+ P

| *<I-cache> <Register file> <D-cache> | r*

| *| o*

| *<Control circuitry> | c*

| *+-+ +-+ +-+ +-+ | e*

| *<f-box> | | <d-box> | | <x-box> | | <m-box> | | <w-box> | s*

| *+-+ +-+ +-+ +-+ | s*

| *f/d d/x x/m m/w | o*

+-------------------------------------------------------------------------+ r

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-cahe. 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 tricky 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")

d

How bad are stalls? Answer: su = ------- . We need to minimize stalls.

1 + s/i

More precisely, the speedup 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)..

1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

l.d f0,0(r1) f d x m w f | d | x | m | w

-

l.d f2,0(r2) f d x m w f | d | x | m | w

- \\_

mul.d f4,f0,f2 f d s x x n w . f | d---x | x | n | w

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'.

1 2 3 4 5 6 7 8 9 0 1 2 1 2 3 4 5 6 7 8 9 0 1 2

mul.d f4,f0,f2 f d x x x x n w f d x x x x n w

mul.d f6,f4,f8 f d s s s x x x x n w . . . f d x x x x n w

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.

1 2 3 4 5 1 2 3 4 5

add r1,r1,8 f d x n w f d x n w

bne r1,r2,label f s d . f d

mul.d f4,f0,f2 f f f f

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.

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 major assumption that all arithmetic functional units, whether

one box or many, are \_fully pipelined\_.

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.

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.

The control signals are too many to permit a brief description. We

therefore invent a set of toy control signals that give us some of the

flavor of control, without getting us bogged down in unnecessary details.

The toy control signals are 'no-op', 'add', 'multiply', 'shift', 'load',

'store', and 'write back'. If any box receives a box-appropriate control

signal at the start of cycle 'c', then it will perform the specified

action during cycle 'c'. The 'no-op' signal means "do nothing".

Note that in Assignment 3, I ignore \_names\_ of control signals, and let all

case analysis be conducted from the knowledge of the \_opcode\_.

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\_ 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. We don't ask that each factor in Moore's Law

increases exponentially; we only ask that their \_product\_ increases

exponentially. Moore's Law, interpreted in this fashion, is alive and well

in the multicore era, with one important qualification.

Update: This statement is no longer true since 2014.

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. A 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.

A picture we have already seen may help:

1 2 3 4 5 6 7 8 9 0 ... clock cycle

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

instr 6 f d x m w

instr 7 f d x m w

... ...

Look at clock cycle 5. We have five boxes working on five different

instructions.

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.

Space-time Diagram

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The figure above is a space-time diagram. These diagrams show the evolution

of the pipeline in time, and are useful in capturing aspects of the control

and data flow. Let me repeat the figure with some actual (but not very

interesting) instructions.

1 2 3 4 5 6 7 8 9

add r1,r2,r3 f d x m w

add r2,r3,r4 f d x m w

add r3,r4,r5 f d x m w

add r4,r5,r6 f d x m w

add r5,r6,r7 f d x m w

The instruction-execution pipeline reaches equilibrium in cycle 5, when all

five boxes are (notionally) active, and five distinct instructions are being

processed at the same time. At equilibrium, one instruction is fetched per

cycle, one result is produced per cycle, the latency is 5 cycles, and the

pipeline occupancy (concurrency) is 5, which is the latency-bandwidth product.

In this toy program, no data flows between instructions, which are all

independent. As described earlier, data does flow between different boxes

working on the same instruction. For example, in the first instruction, the

x-box receives the values of 'r2' and 'r3', which it adds. The w-box receives

the register name "r1" and the sum computed by the x-box. However, the m-box

receives no data at all. Since an add is not a memory reference, there is no

work from the m-box to do. Instead, it performs a no-op. I sometimes write

'n' in place of 'm' when I want to emphasize that the m-box is inactive. The

data sent to the w-box hops right over the m-box.

In this diagram, what control signals do the boxes receive? Consider cycle 1.

The f-box receives the control signal 'fetch', while \_all\_ the other boxes

receive the control signal 'no-op'. By reading the columns of the diagram,

one can see which boxes receive activating control signals, and which boxes

are instructed to do nothing. (Qualification follows).

The exception to this rule is the m-box. Although I have written 'm' in five

columns, in fact, the m-box receives the 'no-op' control signal in \_each and

every\_ cycle (not just the columns in which 'm' appears).

In every cycle, the f- and d-boxes receive either the 'no-op' control signal

or the 'fetch' and 'decode' control signals, respectively. From a control

standpoint, they are not very interesting case statements. For this reason,

we focus on the x-, m-, and w-boxes.

When the x-box does something, it may be any one of a range of arithmetic or

logical operations. When the m-box does something, it is either a load or a

store. When the w-box does something, it is always write back.

A pipeline is a machine with a control system. We will see more interesting

space-time diagrams when we learn about pipeline dynamics. But a space-time

diagram by itself shows the pipeline's kinematics. For all boxes other than

the m-box, if a box name appears in the column corresponding to cycle 'c',

then that box is active in cycle 'c'. Howver, if the m-box's name appears,

then it may or may not be active.

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. Figure C.29 in the 6th edition is

just total nonsense. The authors talk more sense in chapter 3. We will use

the notion of gap in our discussion of \_loop unrolling\_, in chapter 3.

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