Using Data Instead of Branching

Branching in Assembler is a Pain!

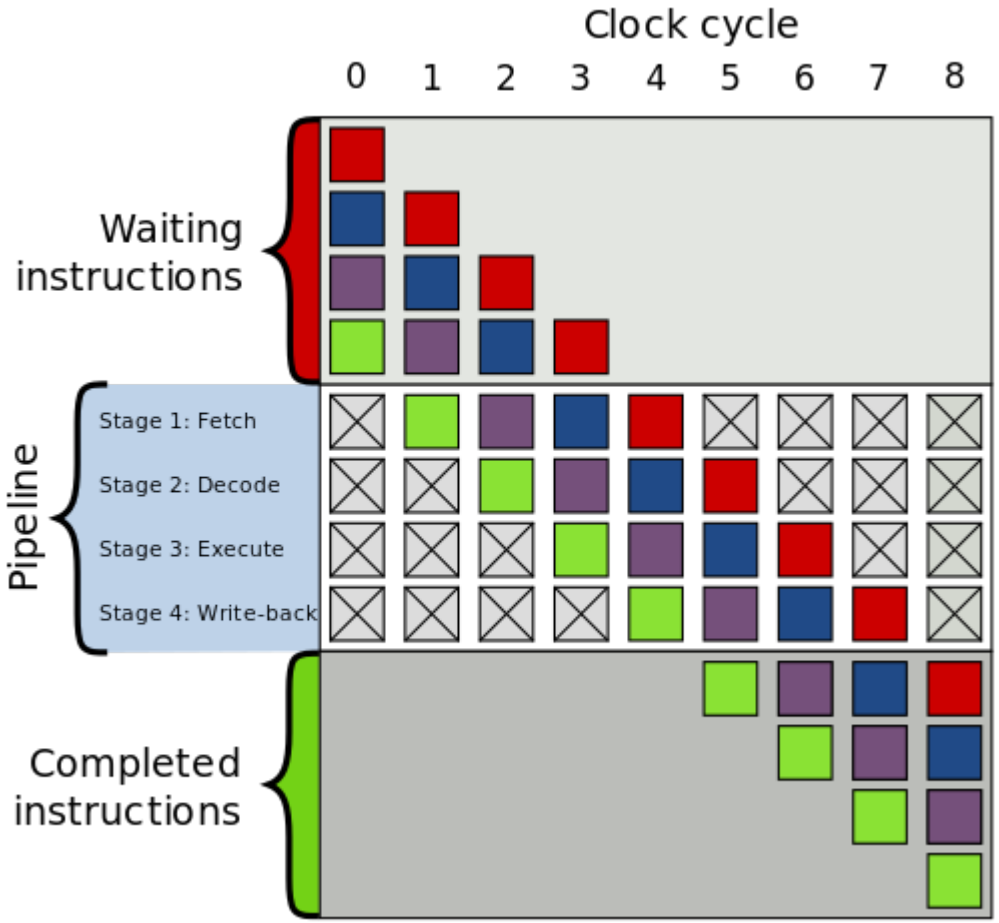
Intro

- Pipelining
- The effect of conditional branching on a pipelined architecture
- Using data to remove conditional branching

Pipelining

Picture from Wikipedia

- 4 stage pipeline shown
- Smaller steps are faster steps
- Increase in latency (more steps)
- Increase execution rate
- Intel pipelines ~10–30 stages



Branching

- Incorrectly predicted branches hurt performance badly in a pipelined architecture
- But we have to make decisions and our code needs to react
- All those jumps give us hard to read “spaghetti code”
- What can we use instead of lots of conditional jumps?

(By now you may be feeling a powerful longing for curly braces and the word “else”)

Let Data Drive the Boat

- Recall the assembler statements homework assignment?
- There's no branching in the following code but it reads like "if something was not equal, increment rax"

```
xorq %rsi, %rsi   #init all the bytes to zero
…
setne %sil        #maybe set the low byte to one
                  #set provides branchless decision!
addq %rsi, %rax   #has no effect if rsi is zero
```

Counting Words Example

- Our counting words example counts transitions from “not in a word” to “in a word” as we progress through a string
- Characters that make up words fall in a range such as a-z
- If the current character falls in the range, it is part of a word
- If the previous character was not in range and the current character is in range, we have found the beginning of a new word.
- We don’t care what the previous character was, but we do care if it was in range

Method for V1 in C – with branches

```
int in_range( char c);
char next_char();

int word_count()
{
        int count = 0;
        int was_not = 1; /* my last char was not in a word */
        char c;

/* while loop on next page */

        return count;
}
```

Method for V1 in C – with branches (loop)

```
while( c = next_char())
{
    if( in_range(c)) /* hide that it takes 2 checks */
    {
        if(was_not) /* am now, but was not before */
        {
            count++;
         }
         was_not = 0; /* for next time */
    }
    else
        was_not = 1; /* for next time */
}
```

Method for V1 in C – fewer branches (loop)

```
while( c = next_char())
{
    if( in_range(c)) /* hide that it take s2 checks */
    {
       count+= was_not;    /* NO BRANCH TO DO THIS! */
       was_not = 0; /* for next time */
    }
    else
        was_not = 1; /* for next time */
}
```

Less Branching When Counting Words (V1)

```
movq $1, %rsi       # re-use rsi.  KEY: Now it holds 1 if previous char
                    #was not in range, 0 if it was
…
#code that will branch away if the current character is not
#in range goes here

#the current character is in a word or we would not be here
addq %rsi, %rax     # if we are already in a word, that adds zero.
                    #If we were not already in a word, that adds 1.
                    #I don't need a final "if" because the numbers
                    #take care of it for me.
xorq   %rsi, %rsi   #before we go on, set previous to indicate we
                    #are now in a word
# go and do the next character
# code not shown for the not-in-a-word code merely sets %rsi back to one
```

Branchless Counting Words

- Can we get rid of even more branching?
- Yes with a few registers, compare, set, and a bit of Boolean logic
- Note that we encode the previous result differently
- Use A, B, and Previous for easier register names

Method for V2, part 1

Current Character is n

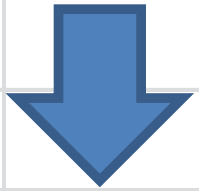
"	n	o	w			i	s	"
P=0	A=1							

P begins at zero, n is in the range so A is 1

A > P

n is the beginning of a word

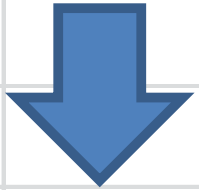
Method for V2, part 2

		Current Character is o								
										
	"	n	o	w			i	s	"	
		P=1	A=1							
		n and o are both in the range, P and A are both 1								
		A is not greater than P								
		o is not the beginning of a word								

Method for V2, part 3

			Current Character is a space							
										
	"	n	o	w			i	s	"	
				P=1	A=0					
	w is in ther range so p=1 but space is not in the range making A=0									
					A is not greater than P					
					space is not the beginning of a word					

Method for V2, part 4

			Current Character is a space							
										
	"	n	o	w			i	s	"	
					P=0	A=0				
	spaces are not in the range, making both P and A zero									
					A is not greater than P					
					space is not the beginning of a word					

Branchless loop body

```
# compare current char to upper bound
# set register A if in current char passes the test (in range)
# compare current char lower bound
# set register B if current char passes that test (in range)
# and those two, result into A
#    (so A is 1 if it passes both tests)
# compare A to Previous (the code looks backwards)
# set register B if greater (when A is 1 and Previous is 0)
# add B to rax
#    (it's 1 when we went from not a word to being in a word)
# assign Previous with the value in A (prev = current)
# time for next character
```

(This encodes the previous result as 0 means not in range)

Branchless loop body – with “code”

```
cmpb %Upper, %C     # compare current char to upper bound
setle %A            # set register A if in current char passes the test
cmpb %Lower, %C     # compare current char lower bound
setge %B            # set register B if current char passes that test (

andb %B, %A         # and those two, result into A
                    #(so A is 1 if it passes both tests)

            # previous = 1 means last character was in range
cmpb %Previous, %A # compare A to Previous (the code looks backwards)
setg %B             # set register B if greater
            # (%B is 1 when we went from not a word to being in a word
            # which is when A is 1 and Previous is 0)

add %B, %rax        # add B to rax (ignore size mismatch)
mov %A, %Previous # assign Previous with the value in A (prev = current)
                    # time for next character
```

C code equivalent

```
lower = (current >= lbound); /* store lower test result */
upper = (current <= ubound); /* store upper test result */
new = lower & upper;     /* combine those tests */
if (new> old) rval++;    /* if current passes and prev did
                         ** not, start of new word -
                         ** count it */
old = new;               /* done here, ready for next */
```

Takeaway (assembler):

- The set instruction can be leveraged to good effect
- Conditional move may also have what you need
- Even without either of those you can use data encoding to reduce branching
  - Adding zero or one
  - Multiply by zero, one, or negative one
  - Using Boolean operations to get these data values

- Data-driven code in high level languages is often more maintainable than procedural code. Data-driven code in assembler can have lower branching.

Takeaway (architecture):

- Deeper pipelines (many stages) execute faster
- Deeper pipelines have a higher cost for a pipeline flush
- Branch prediction mitigates but does not remove this penalty