Machine architecture

January 22, 2019

Contents

BOH 3.1 (page 218)

Assume the following values are stored at the indicated memory addresses and registers.

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Answer

See figure 3.3 on page 217 for help.

Operand	Operand value	Value
%rax	R[%rax]	0x100
0x104	M[0x104]	0xAB
$0x108	Imm	0x108
(%rax)	M[R[%rax]]	0xFF
4(%rax)	M[4 + R[%rax]]	0xAB
9(%rax, %rdx)	M[9 + R[%rax] + R[%rdx]]	0x11
260(%rcx, %rdx)	M[260 + R[%rcx] + R[%rdx]]	0x13
0xFC(, %rcx, 4)	M[0xFC + R[%rcx] * 4]	0xFF
(%rax, %rdx, 4)	M[R[%rax]+R[%rdx] * 4]	0x11

BOH 3.2 (page 221)

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, *mov* can be rewritten as *movb*, *movw*, *movl*, or *movq*).
Before solving the task there is a few rules that is good to remember.

- Everything that is placed on the right side is what we call **D** (destination) and everything on the left side is what we call **S** (Source)
- We always look at the registers size to determine the appropriate instruction suffix.

Useful table for simple movement instructions:

Instruction			Effect	Description
MOV		S, D	D ← S	Move
	movb			Move byte
	movw			Move word
	movl			Move double word
	movq			Move quad word
movabsq		I, R	R ← I	Move absolute quad word

Answer

1. mov**l** %eax, (%rsp)
   From figure 3.2 (page 216) we can see that the register %eax has the size of 4 bytes which we know from the "useful table" can be represented as a double word with the instruction mov**l**.

2. mov**w** (%rax), %dx
   From figure 3.2 (page 216) we can see that the register %dx has the size of 2 bytes which we know from the "useful table" can be represented as a word with the instruction mov**w**.

3. mov**b** $0xFF, %bl
   From figure 3.2 (page 216) we can see that the register %bl has the size of 1 byte which we know from the "useful table" can be represented as a byte with the instruction mov**b**.

4. mov**b** (%rsp, %rdx , 4), %dl
   From figure 3.2 (page 216) we can see that the register %dl has the size of 1 byte which we know from the "useful table" can be represented as a byte with the instruction mov**b**.

5. mov**q** (%rdx), %rdx
   From figure 3.2 (page 216) we can see that the register %rdx has the size of 8 bytes which we know from the "useful table" can be represented as a quad word with the instruction mov**q**.

6. mov**w** %dx, (%rax)
   From figure 3.2 (page 216) we can see that the register %dx has the size of 2 bytes which we know from the "useful table" can be represented as a double word with the instruction mov**w**.

BOH 3.3 (page 222)

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

Answer

1. mov**b** $0xF, (%ebx)
   It is not possible to use memory as a address register.

2. mov**l** %rax, (%rsp)
   Missmatch between the instruction mov**l** which is 4 bytes where the address register %rax is 8 bytes (mov**q**).

3. mov**w** %(rax), 4(%rsp)
   Is is not possible to have memory as both source and destination.

4. mov**b** %al, %sl
   There is no register named %sl.

5. mov**q** %rax, $0x123
   It is now possible to have immediate as destination, you can only have immediate on the left side.

6. mov**l** %eax, %rdx
   Missmatch between the instruction mov**l** which is 4 bytes where the address register %rdx is 8 bytes (mov**q**).

7. mov**b** %si, 8(%rbp)
   Missmatch between the instruction mov**b** which is moving 1 byte but we are actually moving something to a 2 byte register.

BOH 3.5 (page 225)

```
decode1:
    movq   (%rdi), %r8
    movq   (%rsi), %rcx
    movq   (%rdx), %rax
    movq   %r8, (%rsi)
    movq   %rcx, (%rdx)
    movq   %rax, (%rdi)
    ret
```

Answer

xp in %rdi, yp in %rsi, zp in %rdx

```
1  void decode1 (long *xp, long *yp, long *zp) {
2    long x = *xp; // movq (%rdi), %r8
3    long y = *yp; // movq (%rsi), %rcx
4    long z = *zp; // movq (%rdx), %rax
5
6    *yp = x; // movq %r8, (%rsi)
7    *zp = y; // movq %rcx, (%rdx)
8    *xp = z; // movq %rax, (%rdi)
9
10   return; // ret
11 }
```

BOH 3.8 (page 230)

Assume the following values are stored at the indicated memory addresses and registers.

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x108	0xAB	%rcx	0x1
0x110	0x13	%rdx	0x3
0x118	0x11		

Answer

See figure 3.3 on page 217 for help.

Instruction	Operand value	Destination	Computation	Value
addq %rcx,(%rax)	M[R[%rax]]	0x100	0xFF + 0x1	0x100
subq %rdx,8(%rax)	M[8 + R[%rax]]	0x108	0xAB - 0x3	0xA8
imulq $16,(%rax, %rdx, 8)	M[R[%rax] + R[%rdx] * 8]	0x118	0x11 * 16	0x110
incq 16(%rax)	M[16 + R[%rax]]	0x110	0x13+1	0x14
decq %rcx	R[%rcx]	%rcx	0x1 - 1	0x0
subq %rdx, %rax	R[%rax]	%rax	0x100 - 0x3	0xFD

BOH 3.10 (page 232)

```
arith3:
      orq   %rsi, %rdx
      sarq   $9, %rdx
      notq   %rdx
      movq   %rdx, %rbx
      subq   %rsi, %rbx
      ret
```

Answer

x in %rdi, y in %rsi, z in %rdx

```
short arith3(short x, short y, short z) {
  short p1 = y | z;
  short p2 = p1 >> 9;
  short p3 = !p2;
  short p4 = y - p3;
  return p4;
}
```

BOH 3.11 (page 233)

It is common to find assembly-code lines of the form

```
xorq %rcx, %rcx
```

in code that was generated from C where no exclusive-or operations were present

Answer

A. Explain the effect of this particular exclusive - or instruction and what useful operation it implements.

The instruction sets the particular register to 0. $(X \oplus X) = 0$

B. What would be the more straightforward way to express this operation in assembly code?

```
movq $0, %rcx
```

BOH 3.13 (page 240)

See figure 3.13 (p. 238) and 3.14 (p. 239)

```
int comp(data_t a, data_t b) {
      return a COMP b;
}
```

A.

```
cmpl   %esi, %edi
setl   %al

data_t = int
COMP = signed <
```

B.

```
cmpw   %si, %di
setge   %al

data_t = short
COMP = signed >=
```

C.

```
cmpb   %sil, %dil
setbe   %al

data_t = unsigned char
COMP = unsigned <=
```

D.

```
cmpq   %rsi, %rdi
setne   %al

data_t = long or unsigned long
COMP = not equal
```

BOH 3.14 (page 241)

See figure 3.13 (p. 238) and 3.14 (p. 239)

```
int test(data_t a) {
      return a TEST 0;
}
```

A.

```
testq   %rdi, %rdi
setge   %al

data_t = long
TEST = signed >=
```

B.

```
testw   %di, %di
sete    %al

data_t = short or unsigned short
TEST = equal
```

C.

```
testb   %dil, %dil
seta    %al

data_t = unsigned char
TEST = unsigned >
```

D.

```
testl   %edi, %edi
setle   %al

data_t = int
TEST = signed <=
```

BOH 3.18 (page 249)

GCC generates the following assembly code:

```
test
   leaq    (%rdx,%rsi), %rax
   subq    %rdi, %rax
   cmpq    $5, %rdx
   jle     .L2
   cmpq    $2, %rsi
   jle     .L3
   movq    %rdi, %rax
   idivq   %rdx, %rax
   ret
.L3:
   movq    %rdi, %rax
   idivq   %rsi, %rax
   ret
.L2:
   cmpq    $3, %rdx
   jge     .L4
   movq    %rdx, %rax
   idivq   %rsi, %rax
.L4:
   rep; ret
```

Answer

`x in %rdi, y in %rsi, z in %rdx`

```c
short test(short x, short y, short z){
  short val = z + y - x;
  if(z > 5){
    if(y > 2)
      val = x / z;
    else
      val = x / y;
  }else if (z < 3) {
    val = z / y;
  return val;
}
```

BOH 3.23 (page 258)

For the C code

```
1  short dw_loop(short x) {
2    short y = x/9;
3    short *p = &x;
4    short n = 4*x;
5    do {
6      x += y;
7      (*p) += 5;
8      n -= 2;
9    } while (n > 0);
10   return x;
11 }
```

gcc generates the following assembly code:

```
x initially in %rdi

dw_loop:
    movq    %rdi, %rbx
    movq    %rdi, %rcx
    idivq   $9,   %rcx
    leaq    (,%rdi,4), %rdx
.L2:
    leaq    5(%rbx,%rcx), %rcx
    subq    $2, %rdx
    testq   %rdx, %rdx
    jg      .L2
    rep; ret
```

Answer

A. Which registers are used to hold program values x, y, and n?

Although parameter x is passed to the function in register %rdi, we can see that the register is never referenced once the loop is entered. Instead, we can see that registers %rbx, %rcx, and %rdx are initialized in lines 25 to x, x/9, and 4*x. We can conclude, therefore, that these registers contain the program variables.

B. How has the compiler eliminated the need for pointer variable p and the pointer dereferencing implied by the expression $(*p)+=5$?

The compiler determines that pointer p always points to x, and hence the expression $(*p)+=5$ simply increments x. It combines this incrementing by 5 with the increment of y, via the leaq instruction of line 7.

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.19(c).

```
dw_loop:
    movq    %rdi, %rbx              Copy x to %rbx
    movq    %rdi, %rcx
    idivq   $9,   %rcx              Computes y = x/9
    leaq    (,%rdi,4), %rdx         Computes n = 4*x
.L2:
    leaq    5(%rbx,%rcx), %rcx      Computes x += y+5
    subq    $2, %rdx                Decrement n with 2
    testq   %rdx, %rdx              Test n
    jg      .L2                     If n > 0 goto .L2
    rep; ret
```

BOH 3.24 (page 260)

gcc, run with command-line option -Og, produces the following code:
short loop_while(short a, short b)
a in %rdi, b in %rsi

```
loop_while:
  movl   $0, %eax
  jmp    .L2
.L3:
  leaq   (,%rsi, %rdi),  rdx
  addq   %rdx, %rax
  subq   $1, %rdi
.L2:
  cmpq   %rsi, %rdi
  jg.    .L3
  rep; ret
```

Answer

For C code having the general form

```
Short loop-while(short a, short b){
  short result = 0
  while (a > b) {
    result = result + (a * b)
    a = a - 1
  }
  return result;
}
```

BOH 6.7 (page 644)

To create a stride-1 reference pattern, the loops must be permuted so that the rightmost indices change most rapidly.

```
int productarray3d(int a[N][N][N])
{
  int i, j, k, product = 1;

  for (i = N-1; i>= 0; i--) {
    for (j = N-1; j >= 0; j--) {
      for (k = N-1; k >= 0; k--) {
        product *= a[j][k][i];
      }
    }
  }
  return product;
}
```

To create a stride-1 reference patterne we have to change the order of the loops.

Answer

```
int productarray3d(int a[N][N][N])
{
  int i, j, k, product = 1;

  for (j = N-1; j>= 0; j--) {
    for (k = N-1; k >= 0; k--) {
      for (i = N-1; i >= 0; i--) {
        product *= a[j][k][i];
      }
    }
  }
  return product;
}
```

BOH 6.8 (page 645)

(a) An array of structs

```
1   #define N 1000
2
3   typedef struct {
4       int vel[3];
5       int acc[3];
6   } point;
7
8   point p[N];
```

(b) The clear1 function

```
1   void clear1(point *p, int n)
2   {
3       int i, j;
4
5       for (i = 0; i < n; i++) {
6           for (j = 0; j < 3; j++)
7               p[i].vel[j] = 0;
8           for (j = 0; j < 3; j++)
9               p[i].acc[j] = 0;
10      }
11  }
```

(c) The clear2 function

```
1   void clear2(point *p, int n)
2   {
3       int i, j;
4
5       for (i = 0; i < n; i++) {
6           for (j = 0; j < 3; j++) {
7               p[i].vel[j] = 0;
8               p[i].acc[j] = 0;
9           }
10      }
11  }
```

(d) The clear3 function

```
1   void clear3(point *p, int n)
2   {
3       int i, j;
4
5       for (j = 0; j < 3; j++) {
6           for (i = 0; i < n; i++)
7               p[i].vel[j] = 0;
8           for (i = 0; i < n; i++)
9               p[i].acc[j] = 0;
10      }
11  }
```

Figure 6.20 **Code examples for Practice Problem 6.8.**

Practice Problem 6.8 (solution page 699)

The three functions in Figure 6.20 perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

Answer

`clear1` has the best locality because it's accesses the array using a stride-1 reference pattern.

`clear2` has the second best locality because it's scans each of the N structs in order, which is good, but within each struct it hops around in a non-stride-1 pattern.

`clear3` has the worse locality because it's not only hops around within each struct, but also hops from struct to struct.

BOH 6.11 (page 660)

Cache

The problems that follow will help reinforce your understanding of how caches work. Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and eight sets ($S = 8$).

Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

Figure 6.26 **Summary of cache parameters.**

BOH 6.13 (page 664)

Suppose a program running on the machine as described above which references the 1-byte word at address $0x0D53$. Indicate the cache entry accessed and the cache byte value returned in hexadecimal notation. Indixate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned."

Answer

To find the block offset, set index and the tag we use the table on page 653 in BOH.
What we know is that

- B = block size = 4
- S = number of sets = 8
- b = block offset = log(4) = 2
- s = set index bits = log(8) = 3
- number of tag bits:
  t = 13-(3+2)
  t = 13-5 = 8

We can now write the address format, because we know the block offset, set index and tag.
a)

CT								CI			CO	
0	1	1	0	1	0	1	0	1	0	0	1	1

The table above shows the number of block offset, set index and tag bits. The binary number underneath is computed from the address 0x0D53. We now compute the three different offsets to hex and can then answer question b.
b)

Parameter	Value
Cache block offset (CO)	0x3
Cache set index (CI)	0x4
Cache tag (CT)	0x6A
Cache hit? (Y/N)	N
Cache byte returned	—

BOH 6.14 (page 664)

Suppose a program running on the machine as described above which references the 1-byte word at address $0x0CB4$. Indicate the cache entry accessed and the cache byte value returned in hexadecimal notation. Indixate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned."

Answer

To find the block offset, set index and the tag we use the table on page 653 in BOH.
What we know is that

- B = block size = 4
- S = number of sets = 8
- b = block offset = log(4) = 2
- s = set index bits = log(8) = 3
- number of tag bits:
  t = 13-(3+2)
  t = 13-5 = 8

We can now write the address format, because we know the block offset, set index and tag.
a)

CT								CI			CO	
0	1	1	0	0	1	0	1	1	0	1	0	0

The table above shows the number of block offset, set index and tag bits. The binary number underneath is computed from the address 0x0CB4. We now compute the three different offsets to hex and can then answer question b.
b)

Parameter	Value
Cache block offset (CO)	0x0
Cache set index (CI)	0x5
Cache tag (CT)	0x65
Cache hit? (Y/N)	N
Cache byte returned	—

BOH 6.15 (page 664)

Suppose a program running on the machine as described above which references the 1-byte word at address $0x0A31$. Indicate the cache entry accessed and the cache byte value returned in hexadecimal notation. Indixate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned."

Answer

To find the block offset, set index and the tag we use the table on page 653 in BOH.
What we know is that

- B = block size = 4
- S = number of sets = 8
- b = block offset = log(4) = 2
- s = set index bits = log(8) = 3
- number of tag bits:
  t = 13-(3+2)
  t = 13-5 = 8

We can now write the address format, because we know the block offset, set index and tag.
a)

CT								CI			CO	
0	1	0	1	0	0	0	1	1	0	0	0	1

The table above shows the number of block offset, set index and tag bits. The binary number underneath is computed from the address 0x0A31. We now compute the three different offsets to hex and can then answer question b.
b)

Parameter	Value
Cache block offset (CO)	0x1
Cache set index (CI)	0x4
Cache tag (CT)	0x51
Cache hit? (Y/N)	N
Cache byte returned	—

BOH 6.18 (page 673)

Determine the cache performance for the following code:

```
1       for (i = 31; i >= 0; i--) {
2           for (j = 31; j >= 0; j--) {
3               total_x += grid[i][j].x;
4           }
5       }
6
7       for (i = 31; i >= 0; i--) {
8           for (j = 31; j >= 0; j--) {
9               total_y += grid[i][j].y;
10          }
11      }
```

A. What is the total number of reads?

B. What is the total number of reads that miss in the cache?

C. What is the miss rate?

Answer

A.

Because we have four foor-loops where each loop runs from 31-0 we can compute

$$\text{Number of reads: } 32 \cdot 32 = 1024 \cdot 2 = 2048$$

B.

Because we have two integers x and y we can take the number of reads and divide with two, so we now can compute

$$\text{Total number of reads that miss in the cache: } 2048/2$$

C.

$$\text{Miss rate : } \ 1024/2048 = 0.5$$

Which is the same as 50%.

BOH 6.24 (page 685)

Suppose that a 2 MB file consisting of 512-byte logical blocks is stored on disk drive with the following characteristics:

Parameter	Value
Rotational rate	18.000 RPM
$T_{avg\ seek}$	8 ms
Average number of sectors/track	2.000
Surfaces	4
Sector size	512 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first flock is $T_{avg\ seek} + T_{avg\ rotation}$.

A. *Best case:* Estimate the optimal time (in ms) required to read the file given the best possible mapping of logical blocks to disk sectors (i.e., sequential)
B. *Random case:* Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

Answer

Page 630 BOH and practice problem 6.4 solution page 697 is very helpful.

A. Best case

$$\begin{aligned} T_{\text{max rotation}} &= \frac{1}{\text{RPM}} \cdot \frac{60\ sec}{1\ min} \\ &= \frac{60}{18.000} \cdot 1000 \\ &= 3,33 \text{ ms} \end{aligned}$$

$$\begin{aligned} T_{\text{avg rotation}} &= \frac{1}{2} \cdot T_{\text{max rotation}} \\ &= \frac{1}{2} \cdot 3,33 \\ &= 1,665 \text{ ms} \end{aligned}$$

File size 2MB, block size 512B, block count 2MB/512B = 4.000
Block Per Track = 2.000, so we need rotate 2 loop to read all data

$$\begin{aligned} T_{\text{transfer}} &= T_{\text{max rotation}} \cdot 2 \\ &= 3,33 \cdot 2 \\ &= 6,66 \end{aligned}$$

$$\begin{aligned} T_{\text{access}} &= T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{transfer}} \\ &= 8 + 1,665 + 6,66 \\ &= 16,325 \text{ ms} \end{aligned}$$

We can now conclude that the total time to read the file is 16,325 ms.

B. Random case

To compute the time (in ms) required to read the file if blocks are mapped randomly to disk sectors we use the formula

$$\begin{aligned} T_{\text{access}} &= (T_{\text{avg seek}} + T_{\text{avg rotation}}) \cdot \text{block count} \\ &= (8 + 1,665) \cdot 4.000 \\ &= 38.660 \text{ ms} \end{aligned}$$

We can now conclude that the total time to read the file is 38.660 ms.

Boolean arithmetic

Answer

$$\sim (A \ \& \ B) = \sim A \mid \ \sim B$$

A	B	$\sim A$	$\sim B$	$\sim (A\&B)$	$\sim A \mid \sim B$
1	1	0	0	0	0
1	0	0	1	1	1
0	1	1	0	1	1
0	0	1	1	1	1

$$(A \wedge B)\&A = A\& \sim B$$

A	B	$\sim B$	$(A \wedge B)$	$(A \wedge B)\&A$	$A\& \sim B$
1	1	0	0	0	0
1	0	1	1	1	1
0	1	0	1	0	0
0	0	1	0	0	0

$$(A\&B) \mid (C\&(A \wedge B)) = C \wedge ((A \wedge C) \ \& \ (B \wedge C))$$

A	B	C	$(A\&B)$	$(A \wedge B)$	$(C\&(A \wedge B))$	$(A\&B)\|(C\&(A \wedge B))$	$(B \wedge C)$	$(A \wedge C)$	$C \wedge ((A \wedge C)\&(B \wedge C))$
1	1	1	1	0	0	1	0	0	1
1	1	0	1	0	0	1	1	1	1
1	0	1	0	1	1	1	1	0	1
1	0	0	0	1	0	0	0	1	0
0	1	1	0	1	1	1	0	1	1
0	1	0	0	1	0	0	1	0	0
0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	0	0	0

Decompilation

```
p:
    movq    %rdi, %rax
    jmp     .L5
.L3:
    addq    %rsi, %rdx
    addq    $8, %rdi
    movq    %rdx, -8(%rdi)
.L5:
    movq    (%rdi), %rdx
    testq   %rdx, %rdx
    jg      .L3
    subq    %rdi, %rax
    sarq    $3, %rax
    ret
```

Answer

```
long p(long *rdi, long rsi) {
  long *rax = rdi;
  long rdx = *rdi;

  while (rdx > 0) {
    rdx = rsi + rdx
    *rdi++;
    *rdi = rdx;
  }
}
```

Pipeline

Nedenfor ses afviklingen af en stump x86prime kode på en simple pipeline:

Den viste kode er den indre løkke i en funktion som multiplicerer alle elementer i en nul-termineret tabel med et argument.

Ved indgang til løkken indeholder %r10 pointeren til tabellen og %r12 det tal alle elementer skal multipliceres med.

```
Loop:
    movq (%r10),%r11    FDXMYW
    cbe $0,%r11,Done     FDDXMYW
    multq %r12,%r11       FFDXMYW
    movq %r11,(%r10)        FDDDXMYW
    addq $8,%r10             FFFDXMYW
    jmp Loop                    FDXMYW
Done:
```

Bogstaverne til højre viser hver instruktions passage gennem pipelinen. Betydningen af bogstaverne er:

- F: Fetch, instruktionhentning
- D: Decode, afkodning, læsning af registre, evt venten på operander
- X: eXecute, udførelse af ALU op, af adresseberegning eller af første del af multiplikation
- M: Memory, læsning/skrivning af data fra data-cache, midterste del af multiplikation
- Y: sidste del af multiplikation
- W: Writeback, opdatering af registre

Alle instruktioner passerer gennem de samme 6 trin. Multiplikation udføres over 3 pipeline-trin, E, M og Y. Et ubetinget hop udføres i D-trinnet, dvs den instruktion der hoppes til kan blive hentet i cyklussen efter. Et betinget hop udføres derimod først i X-trinnet.

Der er fuld forwarding af operander fra en instruktion til en afhængig instruktion. Instruktioner venter i D-trinnet indtil operander er tilgængelige.

Spg 1:

Vi indfører en særlig undtagelse for movq instruktioner som skriver til lageret. De skal stadig vente i D på operander til adresseberegning, men skal først vente i X på selve den værdi der skal skrives til lageret.

Gentegn figuren ovenfor under hensyntagen til denne ændring.

Spg 2:

Hvor mange clock-cyklusser tager hvert gennemløb af løkken. Udvid eventuelt figuren med instruktioner fra efterfølgende gennemløb, indtil du er sikker på dit svar.

Spg 3:

Kan du optimere koden ved at flytte rundt på instruktionerne således at der er færre instruktioner der skal vente på vej gennem pipelinen?

Vis den optimerede kode og tegn en tilsvarende figur der viser hvordan den udføres i pipelinen. Hvor meget hurtigere er din optimerede kode?

Answer

Spg 1:

```
1     F  D  X  M  Y  W
2        F  D  D  X  M  Y  W
3           F  F  D  X  M  Y  W
4                 F  D  X  X  M  Y  W
5                    F  D  D  X  M  Y  W
6                       F  F  D  X  M  Y  W
```

spg 2:
With the knowledge that a conditional jump happens in the X-step we just count the columns and can see that the clock cycle is 9. Because of the conditional jump in line 6 at X we an already start fetching again at this point.

spg 3:
It can't be optimised.