Processor Architecture

Annalise Tarhan

September 9, 2020

4 Homework Problems

4.45 The x86-64 pushq instruction was described as decrementing the stack pointer and then storing the register at the stack pointer location. This is equivalent to the code sequence below.

```
subq $8, %rsp
movq REG, (%rsp)
```

4.45.1 Does the code sequence correctly describe the behavior of pushq %rsp?

No, it doesn't. When given the instruction pushq %rsp, the original un-decremented value of %rsp is stored on the stack. This version saves the decremented version instead.

4.45.2 How could the code sequence be rewritten to correctly describe the behavior of the instruction for all registers?

```
movq REG, -8(%rsp)
subq $8, (%rsp)
```

4.46 The x86-64 popq instruction was described as copying the result from the top of the stack to the destination register and then incrementing the stack pointer. This is equivalent to the code sequence below.

```
movq (%rsp), REG
addq $8, %rsp
```

4.46.1 Does this code sequence correctly describe the behavior of popq %rsp?

No, it doesn't. When given the instruction popq %rsp, the value popped from the stack is stored at %rsp, not the incremented stack pointer value.

4.46.2 How could the code sequence be rewritten to correctly describe the behavior of the instruction for all registers?

```
addq $8, %rsp
movq -8(%rsp), REG
```

4.47 Consider the following C function, which implements bubblesort using array referencing.

4.47.1 Write and test a C version that references the array elements with pointers, rather than using array indexing.

```
void bubblesort(long *data, long count) {
        long i, last;
        for (last = count-1; last > 0; last--) {
                for (i = 0; i < last; i++) {
                        if (*(data+i+1) < *(data+i)) {
                                long t = *(data+i+1);
                                *(data+i+1) = *(data+i);
                                *(data+i) = t;
                        }
                }
        }
}

int main(int argc, char *argv[]) {
        long nums[5] = {5, 3, 4, 1, 2};
        bubblesort(nums, 5);
        for (int i = 0; i < 5; i++) {
                printf("%lu,", nums[i]);
        }
        return 0;
}
```

4.47.2 Write and test a Y86-64 version.

```
# Execution starts at address 0
        .pos 0
        irmovq stack, %rsp
        call main
        halt

# Array with 5 elements
        .align 8
array:
        .quad 0x5
        .quad 0x3
        .quad 0x1
        .quad 0x2
        .quad 0x4

main:
        irmovq array, %rdi
        irmovq $5, %rsi
        call sort
        ret

# void sort(long *data, long count)

# %rdi  *data   # %r11  &(data+i)
# %rsi  count   # %r12  &(data+i+1)
# %r8   last    # %r13  $1
# %r9   i       # %r14  $8
# %r10  8*i     # %rcx, %rdx temp

sort:
        irmovq $1, %r13         # set constant
        irmovq $8, %r14         # set constant
        rrmovq %rsi, %r8        # last = count
        jmp outer_test

outer_loop:
        irmovq $0, %r9          # i = 0
        irmovq $0, %r10         # 8*i = 0

inner_loop:
        rrmovq %rdi, %r11
        addq %r10, %r11         #  &a[i] = data + 8*i
        rrmovq %rdi, %r12
        addq %r10, %r12
```

```
        addq %r14, %r12          # &a[i+1] = data + 8*i + 8

        mrmovq (%r11), %rcx
        mrmovq (%r12), %rdx
        subq %rcx, %rdx          # temp = a[i+1] - a[i]
        jge inner_test           # if a[i+1] >= a[i], skip

        mrmovq (%r11), %rcx      # temp1 = a[i] (redundant)
        mrmovq (%r12), %rdx      # temp2 = a[i+1]
        rmmovq %rcx, (%r12)      # a[i+1] = temp1
        rmmovq %rdx, (%r11)      # a[i] = temp2

inner_test:
        addq %r13, %r9           # i++
        addq %r14, %r10          # (8*i)++        (+8)
        rrmovq %r8, %rcx         # temp = last
        subq %r9, %rcx           # temp = last - i
        jg inner_loop            # loop if last-i > 0

outer_test:
        subq %r13, %r8           # last--
        jg outer_loop            # loop if last > 0
        ret

# Stack starts here and grows to lower addresses
        .pos 0x200
stack:
```

4.48 Modify bubblesort to implement the test and swap using at most three conditional moves and no jumps.

```
inner_loop:
        rrmovq %rdi, %r11
        addq %r10, %r11          # &a[i] = data+8*i
        rrmovq %rdi, %r12
        addq %r10, %r12
        addq %r14, %r12          # &a[i+1] = data+8*i+8

        mrmovq (%r11), %rcx
        mrmovq (%r12), %rdx
        subq %rcx, %rdx          # temp = a[i+1]-a[i]

        mrmovq (%r11), %rcx      # temp1 = a[i] (redundant)
        mrmovq (%r12), %rdx      # temp2 = a[i+1]
```

```
        cmovl %rcx, %rdx        # COND: temp2 = temp1
        mrmovq (%r12), %rcx     # temp1 = a[i+1]
        rmmovq %rdx, (%r12)     # a[i+1] = temp2
        mrmovq (%r11), %rdx     # temp2 = a[i]
        cmovl %rcx, %rdx        # COND: temp2 = temp1
        rmmovq %rdx, (%r11)     # a[i] = temp2
```

4.49 Modify bubblesort to implement the test and swap using one conditional move and no jumps.

```
inner_loop:
        rrmovq %rdi, %r11
        addq %r10, %r11         # &a[i] = data+8*i
        rrmovq %rdi, %r12
        addq %r10, %r12
        addq %r14, %r12         # &a[i+1] = data+8*i+8

        mrmovq (%r11), %rcx     # temp1 = a[i]
        mrmovq (%r12), %rdx     # temp2 = a[i+1]
        subq %rcx, %rdx         # diff = a[i+1]-a[i]
        irmovq $0, %rcx         # temp1 = 0
        cmovge %rcx, %rdx       # if ordered, diff = 0

        mrmovq (%r11), %rcx     # temp1 = a[i]
        addq %rdx, %rcx         # temp1 = a[i] + diff
        rmmovq %rcx, (%r11)     # a[i] = a[i] + diff

        mrmovq (%r12), %rcx     # temp1 = a[i+1]
        subq %rdx, %rcx         # temp1 = a[i+1] - diff
        rmmovq %rcx, (%r12)     # a[i+1] = a[i+1] - diff
```

4.50 Implement switchv in Y86-64 using a jump table.

```
        .pos 0
        irmovq stack, %rsp
        call main
        halt

# Eight element array: a[0,1,2,3,4,5,6,7]
        .align 8
array:
        .quad 0x0
        .quad 0x1
        .quad 0x2
        .quad 0x3
        .quad 0x4
        .quad 0x5
        .quad 0x6
        .quad 0x7

main:
        irmovq array, %rdi      # &a[0]
        irmovq $0, %rsi         # initial offset
        irmovq $64, %rcx        # end of array
# Useful constants
        irmovq $0, %r10
        irmovq $1, %r11
        irmovq $2, %r12
        irmovq $8, %r13

        call loop
        ret

loop:
# %rdi  &a[0]           # %r10  $0
# %rsi  current offset  # %r11  $1
# %rcx  end of array    # %r12  $2
# %r8   &a[current]     # %r13  $8

        rrmovq %rdi, %r8        # %r8 = &a[0]
        addq %rsi, %r8          # %r8 = &a[current]
        call switchv            # uses %r8 as argument
        rmmovq %rax, (%r8)      # a[current]= result

        addq %r13, %rsi         # increment array offset
        subq %rsi, %rcx         # check if done
        irmovq $64, %rcx        # reset array max
```

```
        jg loop                 # jump if more elements
        ret

# long switchv(long idx)
# %r8   idx
# %r9   jump table address
# %rdx  element
# %r14  temporary values
switchv:
        mrmovq (%r8), %rdx      # %r8 = a[ele]
        irmovq D, %r9           # jump table default

        subq %r10, %rdx         # ele -= 0
        irmovq A, %r14
        cmove  %r14, %r9        # ele == 0

        subq %r12, %rdx         # ele -= 2
        irmovq B, %r14
        cmove %r14, %r9         # ele == 2

        subq %r11, %rdx         # ele -= 1
        irmovq C, %r14
        cmove %r14, %r9         # ele == 3

        subq %r12, %rdx         # ele -= 2
        irmovq B, %r14
        cmove %r14, %r9         # ele == 5

        pushq %r9
        ret

        .pos 0x200
stack:

# Jump Table
A:      irmovq $0xaaa, %rax
        ret
B:      irmovq $0xbbb, %rax
        ret
C:      irmovq $0xccc, %rax
        ret
D:      irmovq $0xddd, %rax
        ret
```

4.51 Describe the computations performed to implement the iaddq instruction.

Instruction: $iaddq\ V, rB$

Fetch: $icode : ifun \leftarrow M_1[PC]$
. $rA : rB \leftarrow M_1[PC + 1]$
. $valC \leftarrow M_8[PC + 2]$
. $valP \leftarrow PC + 10$

Decode: $valB \leftarrow R[rB]$

Execute: $valE \leftarrow valB + valC$
. $SetCC$

Write back: $R[rB] \leftarrow valE$

PC update: $PC \leftarrow valP$

4.52 Modify the HCL descriptions of the control logic blocks in seq-full.hcl to implement the iaddq instruction.

Fetch Stage:

```
bool instr_valid = icode in
{ INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };

bool need_regids =
icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };

bool need_valC =
icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ};
```

Decode Stage:

```
word srcB = [
icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ  } : rB;
... ]

word dstE = [
icode in { IRRMOVQ } && Cnd : rB;
icode in { IIRMOVQ, IOPQ, IIADDQ } : rB;
... ]
```

Execute Stage:

```
word aluA = [
icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : valC;
... ]

word aluB = [
icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
IPUSHQ, IRET, IPOPQ, IIADDQ } : valB;
... ]

bool set_cc = icode in { IOPQ, IIADDQ };
```

4.53 Modify the pipeline control logic so that it correctly handles all possible control and data hazards.

Thanks to dreamanddead:
https://dreamanddead.gitbooks.io/csapp-3e-solutions/content/chapter4/4.53.html

```
# Should I stall or inject a bubble into Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
        # Stalling at fetch when decode needs a value that
        hasn't been written yet
        (d_srcA != RNONE && d_srcA in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM }
        || d_srcB != RNONE && d_srcB in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM }
        # Stalling while ret passes through pipeline
        || IRET in { D_icode, E_icode, M_icode}) &&
        # Don't stall while in a mispredicted branch
        !(E_icode == IJXX && !e_Cnd);

# Should I stall or inject a bubble into Register D?
# At most one of these can be true.
bool D_stall =
        # Stalling at decode when decode needs a value
        that hasn't been written yet
        (d_srcA != RNONE && d_srcA in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM }
        || d_srcB != RNONE && d_srcA in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM })
        # Don't stall while in a mispredicted branch
        && !(E_icode == IJXX && !e_Cnd);
```

```
bool D_bubble =
        # Bubbling while in a mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Bubbling while ret passes through the pipeline
        (IRET in { D_icode, E_icode, M_icode } &&
        # But only when not in a generate/use hazard
        !(d_srcA != RNONE && d_srcA in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM }
        || d_srcB != RNONE && d_srcB in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM}));

# Should I stall or inject a bubble into Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Generate/use hazard
        d_srcA != RNONE && d_srcA in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM }
        || d_srcB != RNONE && d_srcA in {
        E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM };

# Should I stall or inject a bubble into Register M?
# At most one of these can be true.
bool M_stall = 0;
# Start injecting bubbles as soon as exception passes
through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT }
|| W_stat in { SADR, SINS, SHLT };

# Should I stall or inject a bubble into Register W?
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;
```

4.54 Modify pipe-full.hcl to implement the iaddq instruction.

Fetch Stage:

```
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
        IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };
```

```
bool need_regids =
        f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };
```

```
bool need_valC = f_icode in
        {IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ};
```

Decode Stage:

```
word d_srcB = [
D_icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ  } : D_rB;
... ];
```

```
word d_dstE = [
D_icode in { IRRMOVQ, IIRMOVQ, IOPQ, IIADDQ } : D_rB;
... ];
```

4.55 Modify the branch prediction logic in pipe-nt.hcl to predict conditional jumps as not being taken while continuing to predict unconditional jumps and call as being taken.

The new constant in the starter code was misnamed and the guiding comments were misleading. (They were supposed to indicate which parts of the code needed to be changed, but half of them were missing.) There's no way I could have done this without a peek at the solutions manual.
Fetch Stage:

```
word f_pc = [
M_icode == IJXX && M_ifun != UNCOND && M_Cnd : M_valE;
... ];
```

```
word f_predPC = [
f_icode == ICALL ||
(f_icode == IJXX && f_ifun == UNCOND) : f_valC;
... ];
```

Execute Stage:

```
word aluA = [
E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX } : E_valC;
... ];
```

```
word aluB = [
E_icode in { IRRMOVQ, IIRMOVQ, IJXX } : 0;
... ];
```

Pipeline Register Control:

```
bool D_bubble =
(E_icode == IJXX && E_ifun != UNCOND && e_Cnd) ||
... ;
```

```
bool E_bubble =
(E_icode == IJXX && E_ifun != UNCOND && e_Cnd) ||
... ];
```

4.56 Modify the branch prediction logic in pipe-btfnt.hcl so that it predicts conditional jumps as being taken when $valC < valP$ and not taken otherwise. Continue to predict unconditional jumps and call as being taken.

Fetch Stage:

```
word f_pc = [
M_icode == IJXX && M_ifun != UNCOND &&
        M_valE >= M_valA && M_Cnd : M_valE;
M_icode == IJXX && M_ifun != UNCOND &&
        M_valE < M_valA && !M_Cnd : M_valA;
... ];
```

```
word f_predPC = [
f_icode == IJXX &&
        (f_ifun == UNCOND || f_valC < f_valP) : f_valC;
f_icode in { ICALL } : f_valC;
1 : f_valP;
];
```

Execute Stage:

```
word aluA = [
E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX } : E_valC;
... ];
```

```
word aluB = [
E_icode in { IRRMOVQ, IIRMOVQ, IJXX } : 0;
... ];
```

Pipeline Control Register:

```
bool D_bubble =
(E_icode == IJXX && E_ifun != UNCOND &&
(E_valC < E_valA && !e_Cnd || E_valC >= E_valA && e_Cnd))
|| ... ;
```

```
bool E_bubble =
(E_icode == IJXX && E_ifun != UNCOND &&
(E_valC < E_valA && !e_Cnd || E_valC >= E_valA && e_Cnd))
|| ... ;
```

4.57 Load Forwarding

4.57.1 Write a logic formula describing the detection condition for a load/use hazard, except that it will not cause a stall in cases where load forwarding can be used.

$E\_icode \in \{IMRMOVQ, IPOPQ\}$ && $E\_dstM \in \{d\_srcA, d\_srcB\}$ && $D\_icode \notin \{IRMMOVQ, IPUSHQ\}$

4.57.2 Modify pipe-lf.hcl to implement load forwarding.

Execute Stage:

```
word e_valA = [
M_dstM == E_srcA && E_icode in {IPUSHL, IRMMOVL} : m_valM;
1 : E_valA;
];
```

Pipeline Register Control:

```
bool F_stall =
E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB}
&& !(D_icode in IRMMOVQ, IPUSHQ) || ... ;
```

```
bool D_stall =
E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB}
&& !(D_icode in IRMMOVQ, IPUSHQ);
```

```
bool D_bubble =
!(E_icode in {IMRMOVQ, IPOPQ} &&
E_dstM in {d_srcA, d_srcB} &&
!(D_icode in IRMMOVQ, IPUSHQ)) || ... ;
```

4.58 Modify the control logic in pipe-1w.hcl to process popq instructions using only one register port.

Fetch Stage:

```
word f_icode = [
imem_error : INOP;
imem_icode == IPOPQ && D_icode == IPOPQ : IPOP2;
... ];
```

```
bool instr_valid = f_icode in
{ INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IPOP2 };
```

Note: solution says IPOPQ needs a reg id, but I just don't think so.

```
bool need_regids =
f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOP2,
IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

```
word f_predPC = [
f_icode in { IPOPQ } : f_pc;
... ];
```

Decode Stage:

```
word d_srcA = [
D_icode in { IRET, IPOPQ } : RRSP;
... ];
```

```
word d_srcB = [
D_icode in { IPUSHQ, ICALL, IRET, IPOPQ, IPOP2 } : RRSP;
... ];
```

```
word d_dstE = [
D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
... ];
```

```
word d_dstM = [
D_icode in { IMRMOVQ, IPOP2 } : D_rA;
... ];
```

Execute Stage:

```
word aluA = [
E_icode in { ICALL, IPUSHQ, IPOP2 } : -8;
E_icode in { IRET, IPOPQ } : 8;
... ];
```

```
word aluB = [
E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
IPUSHQ, IRET, IPOPQ, IPOP2 } : E_valB;
... ];
```

Memory Stage:

```
word mem_addr = [
M_icode in
{ IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ, IPOP2 } : M_valE;
... ];
```

```
bool mem_read = M_icode in { IMRMOVQ, IPOP2, IRET };
```

Pipeline Register Control:

```
bool D_bubble =
!(E_icode in { IMRMOVQ, IPOP2 } && ... ;
```

```
bool E_bubble =
E_icode in { IMRMOVQ, IPOP2 } && ... ;
```