4.47

A

```
void bubble_ptr(long *data, long count)
{
    for (long *last = data + count - 1; last > data; last--)
    {
        for (long *first = data; first < last; first++)
        {
            if (first[1] < *first)
            {
                long t = first[1];
                first[1] = *first;
                *first = t;
            }
        }
    }
}
```

B

```
bubble_ptr:
    pushq %rbp
    ;%rdi:data
    ;%rsi:count
    irmovq $1,%rax
    subq %rax,%rsi;count-1
    addq %rsi,%rsi;2(count-1)
    addq %rsi,%rsi;4(count-1)
    addq %rsi,%rsi;8(count-1)
    rrmovq %rdi %rax
    addq %rax,%rsi;%rsi=last
    irmovq $8,%rcx
    jmp .L10
.L11:
    addq    %rcx, %rax
.L13:
    rrmovq  %rsi,%rdx
    subq    %rax, %rdx
    jle .L15
    mrmovq  8(%rax), %rdx;%rdx=first[1]
    mrmovq  (%rax), %r8;%r8=*first
    rrmovq  %rdx,%r9
    subq    %r8, %r9
    jge .L11
    rmmovq  %r8, 8(%rax)
    rmmovq  %rdx, (%rax)
    jmp .L11
.L15:
    subq    %rcx, %rsi;last--
```

```
.L10:
    rrmovq  %rsi,%rdx
    subq    %rdi, %rdx
    jle .L16
    rrmovq  %rdi, %rax;%rax=first
    jmp .L13
.L16:
    ret
```

4.51

iaddq

	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	$valE \leftarrow valB + valC$
..	
Memory	
WriteBack	$R[rB] \leftarrow valE$
..	
PC	$PC \leftarrow valP$

4.55

```
#/* $begin pipe-all-hcl */
##################################################################
#    HCL Description of Control for Pipelined Y86 Processor      #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002   #
##################################################################

## Your task is to modify the design so that conditional branches are
## predicted as being not-taken.  The code here is nearly identical
## to that for the normal pipeline.
## Comments starting with keyword "BNT" have been added at places
## relevant to the exercise.

##################################################################
#    C Include's.  Don't alter these                             #
##################################################################
```

```
16
17  quote '#include <stdio.h>'
18  quote '#include "isa.h"'
19  quote '#include "pipeline.h"'
20  quote '#include "stages.h"'
21  quote '#include "sim.h"'
22  quote 'int sim_main(int argc, char *argv[]);'
23  quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
24
25  ##################################################################
26  #    Declarations.  Do not change/remove/delete any of these     #
27  ##################################################################
28
29  ##### Symbolic representation of Y86 Instruction Codes #############
30  intsig INOP     'I_NOP'
31  intsig IHALT    'I_HALT'
32  intsig IRRMOVL  'I_RRMOVL'
33  intsig IIRMOVL  'I_IRMOVL'
34  intsig IRMMOVL  'I_RMMOVL'
35  intsig IMRMOVL  'I_MRMOVL'
36  intsig IOPL 'I_ALU'
37  intsig IJXX 'I_JMP'
38  intsig ICALL    'I_CALL'
39  intsig IRET 'I_RET'
40  intsig IPUSHL   'I_PUSHL'
41  intsig IPOPL    'I_POPL'
42
43  ##### Symbolic representation of Y86 Registers referenced explicitly #####
44  intsig RESP     'REG_ESP'       # Stack Pointer
45  intsig RNONE    'REG_NONE'      # Special value indicating "no register"
46
47  ##### ALU Functions referenced explicitly ##########################
48  intsig ALUADD   'A_ADD'     # ALU should add its arguments
49  ## BNT: For modified branch prediction, need to distinguish
50  ## conditional vs. unconditional branches
51  ##### Jump conditions referenced explicitly
52  intsig JUNCOND 'J_YES'          # Code for unconditional jump instruction
53
54  ##### Signals that can be referenced by control logic ##############
55
56  ##### Pipeline Register F ##########################################
57
58  intsig F_predPC 'pc_curr->pc'       # Predicted value of PC
59
60  ##### Intermediate Values in Fetch Stage ###########################
61
62  intsig f_icode  'if_id_next->icode'  # Fetched instruction code
63  intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
64  intsig f_valC   'if_id_next->valc'   # Constant data of fetched
    instruction
65  intsig f_valP   'if_id_next->valp'   # Address of following instruction
66
67  ##### Pipeline Register D ##########################################
68  intsig D_icode 'if_id_curr->icode'  # Instruction code
69  intsig D_rA 'if_id_curr->ra'    # rA field from instruction
70  intsig D_rB 'if_id_curr->rb'    # rB field from instruction
71  intsig D_valP 'if_id_curr->valp'    # Incremented PC
72
```

```
73  ##### Intermediate Values in Decode Stage  ########################
74
75  intsig d_srcA    'id_ex_next->srca' # srcA from decoded instruction
76  intsig d_srcB    'id_ex_next->srcb' # srcB from decoded instruction
77  intsig d_rvalA 'd_regvala'      # valA read from register file
78  intsig d_rvalB 'd_regvalb'      # valB read from register file
79
80  ##### Pipeline Register E ########################################
81  intsig E_icode 'id_ex_curr->icode'  # Instruction code
82  intsig E_ifun  'id_ex_curr->ifun'       # Instruction function
83  intsig E_valC  'id_ex_curr->valc'   # Constant data
84  intsig E_srcA  'id_ex_curr->srca'       # Source A register ID
85  intsig E_valA  'id_ex_curr->vala'       # Source A value
86  intsig E_srcB  'id_ex_curr->srcb'       # Source B register ID
87  intsig E_valB  'id_ex_curr->valb'       # Source B value
88  intsig E_dstE 'id_ex_curr->deste'   # Destination E register ID
89  intsig E_dstM 'id_ex_curr->destm'   # Destination M register ID
90
91  ##### Intermediate Values in Execute Stage ########################
92  intsig e_valE 'ex_mem_next->vale'   # valE generated by ALU
93  boolsig e_Bch 'ex_mem_next->takebranch' # Am I about to branch?
94
95  ##### Pipeline Register M                  #####
96  intsig M_icode 'ex_mem_curr->icode' # Instruction code
97  intsig M_ifun  'ex_mem_curr->ifun'  # Instruction function
98  intsig M_valA  'ex_mem_curr->vala'      # Source A value
99  intsig M_dstE 'ex_mem_curr->deste'  # Destination E register ID
100 intsig M_valE  'ex_mem_curr->vale'      # ALU E value
101 intsig M_dstM 'ex_mem_curr->destm'  # Destination M register ID
102 boolsig M_Bch 'ex_mem_curr->takebranch' # Branch Taken flag
103
104 ##### Intermediate Values in Memory Stage ##########################
105 intsig m_valM 'mem_wb_next->valm'   # valM generated by memory
106
107 ##### Pipeline Register W ########################################
108 intsig W_icode 'mem_wb_curr->icode' # Instruction code
109 intsig W_dstE 'mem_wb_curr->deste'  # Destination E register ID
110 intsig W_valE  'mem_wb_curr->vale'      # ALU E value
111 intsig W_dstM 'mem_wb_curr->destm'  # Destination M register ID
112 intsig W_valM  'mem_wb_curr->valm'  # Memory M value
113
114 ####################################################################
115 #    Control Signal Definitions.                                   #
116 ####################################################################
117
118 ################ Fetch Stage     ###################################
119
120 ## What address should instruction be fetched at
121 int f_pc = [
122     # Mispredicted branch.  Fetch at incremented PC
123
124     #Modified:When command is jxx(not jmp), if mispredicted branch chosen
    jump to valA
125     M_icode == IJXX && E_ifun != UNCOND && M_Bch : M_valA;
126
127     # Completion of RET instruction.
128     W_icode == IRET : W_valM;
129     # Default: Use predicted value of PC
```

```
    1 : F_predPC;
];

# Does fetched instruction require a regid byte?
bool need_regids =
    f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                 IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
    f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

bool instr_valid = f_icode in
    { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

# Predict next value of PC
int new_F_predPC = [
    # BNT: This is where you'll change the branch prediction rule

    #Added:When command is jxx(not jmp), firstly do not jump
    M_icode == IJXX && E_ifun != UNCOND:f_valP

    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];


################ Decode Stage ######################################


## What register should be used as the A source?
int new_E_srcA = [
    D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
    D_icode in { IPOPL, IRET } : RESP;
    1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int new_E_srcB = [
    D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
    D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
    D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
    D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE;  # Don't need register
];

## What register should be used as the M destination?
int new_E_dstM = [
D_icode in { IMRMOVL, IPOPL } : D_rA;
    1 : RNONE;  # Don't need register
];

```

```
## What should be the A value?
## Forward into decode stage for valA
int new_E_valA = [

    D_icode == IJXX && E_ifun != UNCOND:E_valC;
    D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
    d_srcA == E_dstE : e_valE;    # Forward valE from execute
    d_srcA == M_dstM : m_valM;    # Forward valM from memory
    d_srcA == M_dstE : M_valE;    # Forward valE from memory
    d_srcA == W_dstM : W_valM;    # Forward valM from write back
    d_srcA == W_dstE : W_valE;    # Forward valE from write back
    1 : d_rvalA;  # Use value read from register file
];

int new_E_valB = [
    d_srcB == E_dstE : e_valE;    # Forward valE from execute
    d_srcB == M_dstM : m_valM;    # Forward valM from memory
    d_srcB == M_dstE : M_valE;    # Forward valE from memory
    d_srcB == W_dstM : W_valM;    # Forward valM from write back
    d_srcB == W_dstE : W_valE;    # Forward valE from write back
    1 : d_rvalB;  # Use value read from register file
];

################ Execute Stage #####################################

# BNT: When some branches are predicted as not-taken, you need some
# way to get valC into pipeline register M, so that
# you can correct for a mispredicted branch.

## Select input A to ALU
int aluA = [
    E_icode in { IRRMOVL, IOPL } : E_valA;
    E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
    E_icode in { ICALL, IPUSHL } : -4;
    E_icode in { IRET, IPOPL } : 4;
    # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
    E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                 IPUSHL, IRET, IPOPL } : E_valB;
    E_icode in { IRRMOVL, IIRMOVL } : 0;
    # Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
    E_icode == IOPL : E_ifun;
    1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode == IOPL;


################ Memory Stage ######################################

```

```
## Select memory address
int mem_addr = [
M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
    M_icode in { IPOPL, IRET } : M_valA;
    # Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };


################ Pipeline Register Control #########################

# Should I stall or inject a bubble into Pipeline Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
     E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch

    (E_icode == IJXX && E_ifun != UNCOND && e_Bch) ||
    # Stalling at fetch while ret passes through pipeline
    # but not condition for a load/use hazard
    !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
      IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
    # Mispredicted branch

    (E_icode == IJXX && E_ifun != UNCOND && e_Bch) ||
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
       E_dstM in { d_srcA, d_srcB};

# Should I stall or inject a bubble into Pipeline Register M?
# At most one of these can be true.
bool M_stall = 0;
bool M_bubble = 0;
#/* $end pipe-all-hcl */
```

4.59

4.47的效果更佳。因为4.47拥有更短的指令长度，因此在尽管存在分支，可能出现预测错误，但是却因为代码长度更短，指令数更少而速度优于其它两者。