第二章 信息的表示和处理

Unsigned Arithmetic

Operation	Overflow Condition	Overflow Detection
Addition	$2^w \le x + y \le 2^{w+1} - 2$	x+y <x
Multiplication	$2^w \le x * y \le (2^{w-1})^2$	x && (x*y)/x != y

Signed Arithmetic

Operation	Overflow Condition	Overflow Detection
Addition	$2^{w-1} \le x + y$ OR $x + y < -2^{w-1}$	(x>0&& y>0 && x+y<0) || (x<0 && y<0&& x+y>=0)
Multiplication	$2^{w-1} \le x * y$ OR $x * y < -2^{w-1}$	x && (x*y)/x != y

IEEE Floating Point Representation

- Smallest positive unnormalized value: $2^{-n-2^{k-1}+2}$
- Largest positive unnormalized value: $(1-2^{-n}) \times 2^{-2^{k-1}+2}$
- Smallest positive normalized value: $2^{-2^{k-1}+2}$
- Largest positive normalized value: $(2-2^{-n}) \times 2^{2^{k-1}-1}$

第三章 程序的机器级表示

Registers

- Six registers used in parameter passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Caller-save registers: %rax, %rcx, %rdx, %rdi, %rsi, %r8, %r9, %r10, %r11
- Callee-save registers: %rbx, %rbp, %r12, %r13, %r14, %r15
- Floating point registers: %ymm0~%ymm15, all caller-saved, 256 bits
- Register names

```
%eax    %ax     %al
%ebx    %bx     %bl
%ecx    %cx     %cl
%edx    %dx     %dl
%esi    %six    %sil
%edi    %dix    %dil
%ebp    %bp     %bpl
%esp    %sp     %spl
%r8d    %r8w    %r8b    # 9~15 likewise
```

Machine Instructions

- `movq` accepts 32-bit immediate only and performs signed extension. `movabsq` writes 64-bit immediate to register (only register!).
- `movzlq` does not exist since the processor sets the high 32 bits to zero when copying to the low 32 bits. `movslq` exists. `cltq` sign-extends %eax to %rax, `clto` sign-extends %rax to %rdx:%rax. All move instructions with extension accepts register / memory address as source, and register as destination.
- Unary and binary arithmetic / logical operations accepts immediate only at operand 1.
- Shift operations accepts only immediate / %cl as operand 1.
- `imulq`, `mulq`, `idivq`, `divq`

```
imulq   S   # %rdx:%rax = S * %rax  (After truncation they're the same,
mulq    S   # %rdx:%rax = S * %rax   but not before)
idivq   S   # %rax = %rdx:%rax / S
            # %rdx = %rdx:%rax % S
divq    S   # %rax = %rdx:%rax / S
            # %rdx = %rdx:%rax % S
```

- Condition codes in addition `t=a+b`

```
bool CF = (unsigned) t < (unsigned) a;
bool ZF = t == 0;
bool SF = t < 0;
bool OF = (a<0 == b<0) && (t<0 != a<0);
```

- Destination of set instructions can only be single-byte registers.

Floating point instructions

```
vmovss          S, D    # move single float
vmovsd          S, D    # move single double
vmovaps         S, D    # move aligned packed floats
vmovapd         S, D    # move aligned packed doubles

vcvttss2si(q)   S, D    # convert and truncate float to int/long
vcvttsd2si(q)   S, D    # convert and truncate double to int/long
vcvtsi2ss(q)    S, X, X # convert int/long to float
vcvtsi2sd(q)    S, X, X # convert int/long to double

vunpcklps       X, X, X # unpack and interleave low bits of packed floats
vcvtps2pd       X, X    # convert packed floats to doubles

vmovddup        X, X    # move low bits of doubles to high
vcvtpd2psx      X, X    # convert packed doubles to floats

vaddss          S, R, R # operand 2 and destimation must be a register
vsubss          S, R, R
vmulss          S, R, R
vdivss          S, R, R
vmaxss          S, R, R
```

```
vminss          S, R, R
sqrtss          S, R, R
vxorps          S, R, R
vandps          S, R, R

ucomiss         S, R    # if NaN, sets PF; operand 2 must be a register
                        # use jbe or jp after comparison
```

第四章 处理器体系结构

Implementation of Machine Instructions

		OP rA, rB	rrmov/cmovXX rA, rB	irmov V, rB	rmmov rA, D(rB)	mrmov D(rB), rA	jXX Dest	call Dest	push rA	ret	pop rA	iadd V, rB	leave
Fetch	icode:ifun	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]	M_1[PC]
	rA:rB	M_1[PC+1]	M_1[PC+1]	M_1[PC+1]	M_1[PC+1]	M_1[PC+1]			M_1[PC+1]		M_1[PC+1]	M_1[PC+1]	
	valC			M_8[PC+2]	M_8[PC+2]	M_8[PC+2]	M_8[PC+2]	M_8[PC+2]				M_8[PC+2]	
	valP	PC+2	PC+2	PC+10	PC+10	PC+10	PC+9	PC+9	PC+2	PC+1	PC+2	PC+10	PC+1
Decode	valA	R[rA]	R[rA]		R[rA]	R[rA]			R[rA]	R[%rsp]	R[%rsp]		R[%rbp]
	valB	R[rB]			R[rB]	R[rB]		R[%rsp]	R[%rsp]	R[%rsp]	R[%rsp]	R[rB]	R[%rbp]
Execute	valE	valB OP valA		0+valC	valB+valC	valB+valC		valB+(–8)	valB+(–8)	valB+8	valB+8	valB+valC	valB+8
	cnd		Cond(CC, ifun)				Cond(CC, ifun)						
Memory	M_8[valE]				valA			valP	valA				
	valM					M_8[valE]				M_8[valA]	M_8[valA]		M_8[valA]
Writeback	R[%rsp]							valE	valE	valE	valE		valE
	R[rA]					valM					valM		
	R[rB]	valE	valE if cnd	valE								valE	R[%rbp] = valM
PC Updt.	PC	valP	valP	valP	valP	valP	cnd?valC:valP	valC	valP	valM	valP	valP	valP

- `valE` is always the address for memory access except it is used to update %rsp.
- %rsp is updated in call, push, ret, pop & leave.
- `rA` = %rsp in ret & pop.
- `rB` = %rsp in call, push, ret & pop.
- Memory write in call, push, rmmov; memory read in ret, pop, mrmov.

Important HCL Code

```
## What address should instruction be fetched at
int f_pc = [
    # Mispredicted branch.  Fetch at incremented PC
    M_icode == IJXX && !M_Cnd : M_valA;
    # Completion of RET instruction.
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];

# Predict next value of PC
int f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];

## What register should be used as the A source?
int d_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 d_srcB = [
    D_icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL  } : 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 d_dstE = [
    D_icode in { IRRMOVL, IIRMOVL, IOPL, IIADDL } : D_rB;
    D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE;  # Don't write any register
];

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

## What should be the A value?
## Forward into decode stage for valA
int d_valA = [
    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 d_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
];

## Select input A to ALU
int aluA = [
    E_icode in { IRRMOVL, IOPL } : E_valA;
    E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : 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, IIADDL } : E_valB;
```

```
    E_icode in { IRRMOVL, IIRMOVL } : 0;
    # Other instructions don't need ALU
];

## Should the condition codes be updated?
bool set_cc = E_icode in { IOPL, IIADDL } &&
    # State changes only during normal operation
    !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };

## Set dstE to RNONE in event of not-taken conditional move
int e_dstE = [
    E_icode == IRRMOVL && !e_Cnd : RNONE;
    1 : E_dstE;
];

## 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 };

# 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_Cnd) ||
    # 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_Cnd) ||
    # 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;
# 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 Pipeline Register W?
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;
#/* $end pipe-all-hcl */
```

What's in pipeline registers?

```
F   predPC
D   stat    icode:ifun  rA:rB   valC    valP
E   stat    icode:ifun  valA    valC    valB    dstE    dstM    srcA    srcB
M   stat    icode:ifun  valA    valE    Cnd     dstE    dstM
W   stat    icode:ifun  valA    valE            dstE    dstM
```

Exception Handling

- ret + load/use: D_stall only, no D_bubble

```
return              IRET in {D_icode, E_icode, M_icode}     F_stall && D_bubble
mispred. branch     E_icode == IJXX && !e_Cnd               D_bubble && E_bubble
load/use hazard     E_icode in {IMRMOVL, IPOPL} &&          F_stall && D_stall
                        E_dstM in {d_srcA, d_srcB}              && E_bubble
error               m_stat in {SADR, SINS, SHLT} ||         M_bubble && not_set_cc
                        W_stat in {SADR, SINS, SHLT}
```

第六章 储存器层次结构

DRAM (Dynamic Random-Access Memory)

- A DRAM with d supercells and w DRAM units each supercell stores $d \times w$ bits of information. Usually $w = 8$.
- Supercells are organized into an $r \times c$ array. Each supercell has an address (i, j).

- The DRAM I/O chip consists of 2 address pins and 8 data pins. During memory access, Row Access Strobe request is sent from the memory controller through the address pins, and the chip moves the corresponding row to internal row buffer. When the CAS request is sent, the chip sends the contents of supercell (i, j) back to the controller.
- Multiple DRAMs form memory modules, which together with the memory controller, build up the main memory.
- FPM DRAM (VRAM) -> EDO DRAM -> SDRAM -> DDR SDRAM

  FPM: Fast Page Mode

  EDO: Extended Data Out

  DDR S: Double Data-Rate Synchronous

Nonvolatile Memories, ROMs

- PROM -> EPROM -> EEPROM (Electrically Erasable Programmable Read Only Memory)

Cache

- Different kinds of cache miss: cold miss, conflict miss, capacity miss (p. 424).
- Cache visit time (cycles): L1: 4, L2: 10, L3: 50.
- $m = t + s + b$, where m is memory address length, t is tag length, 2^s is the number of sets and 2^b is block size.
- Write through -> Not-Write-Allocate (Simple to implement).

  Write back -> Write-Allocate (keep data at low level caches).