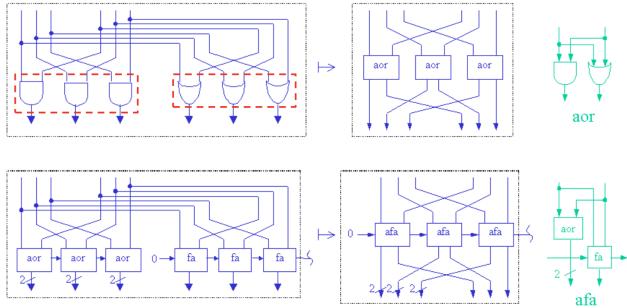


ALU

Deriving ALU cell by interleaving components

- group components together to form larger repeated unit



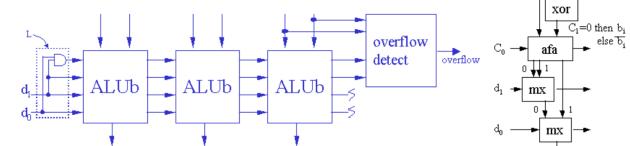
- the dotted boxes have the same function and interface

wl 2022 4.8

Selecting ALU operation

- programmable inverter for b_i (using xor)

- connecting mux in series



- $d_0 d_1: 00$ and, 01 or, 10 add, 11 subtract

- detecting overflow: exercise

wl 2022 4.9

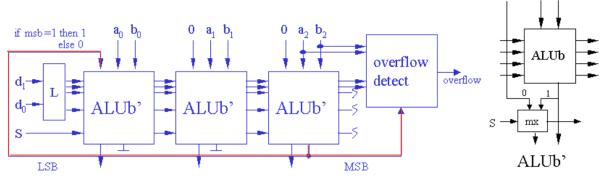
Comparison operations

- slt: set on less than, if $a < b$ then 1 else 0

- if $a < b$, $a - b < 0$, so MSB of $(a - b)$ is 1 (32 bits)

- implementation

- provide additional input to each cell, left of a_i, b_i
- LSB input from MSB ALUb output, other inputs set to 0
- include additional mux in cell for selection
- to select slt, $s=0, d_0=1, d_1=1$

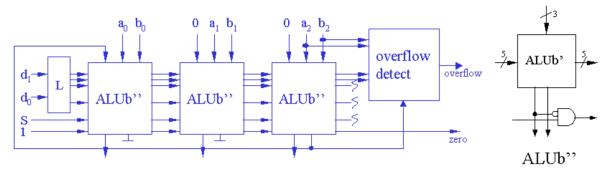


wl 2022 4.10

Zero detection

- beq, bne: test $a=b$ or $a-b=0$

- include another gate to test if output zero



wl 2022 4.11

Multiplication and Division

Comparing the third algorithm and Booth's algorithm for positive numbers

Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial Values	0000 0110 0	Initial Values	0000 0110 0
1	0010 0010	1: 0 => no operation 2: Shift right Product	0000 0110 0	1a: 00 => no operation 2: Shift right Product	0000 0110 0
2	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0010 0011 0	1c: 10 => Prod=Prod-Mcand 2: Shift right Product	1110 0011 0
3	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0011 0001 1	1d: 11 => no operation 2: Shift right Product	1111 0001 1
4	0010 0010	1: 0 => no operation 2: Shift right Product	0001 1000 1	1b: 01 => Prod=Prod+Mcand 2: Shift right Product	0001 1000 1

case {
00: middle of a string of 0s, no action
01: end of a string of 1s, pr = pr + shifted mc
10: start of a string of 1s, pr = pr - shifted mc
11: middle of a string of 1s, no action
then shift right pr}

wl 2022 5.15

Division example using first algorithm

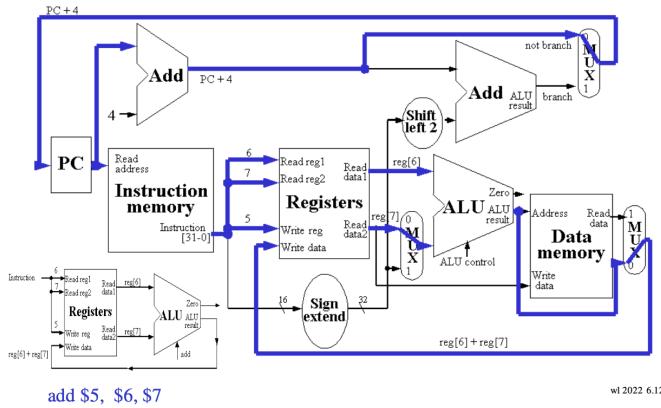
Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div 2b: Rem<0 => +Div, sll Q, Q0=0 3: Shift Div right	0000	0010 0000	1110 0111
2	1: Rem = Rem - Div 2b: Rem<0 => +Div, sll Q, Q0=0 3: Shift Div right	0000	0010 0000	0000 0111
3	1: Rem = Rem - Div 2b: Rem<0 => +Div, sll Q, Q0=0 3: Shift Div right	0000	0001 0000	0000 0111
4	1: Rem = Rem - Div 2a: Rem<0 => sll Q, Q0=1 3: Shift Div right	0000	0000 1000	1111 1111
5	1: Rem = Rem - Div 2a: Rem<0 => sll Q, Q0=1 3: Shift Div right	0001	0000 0010	0000 0001

dd=7 q=3
ds=2 r=1

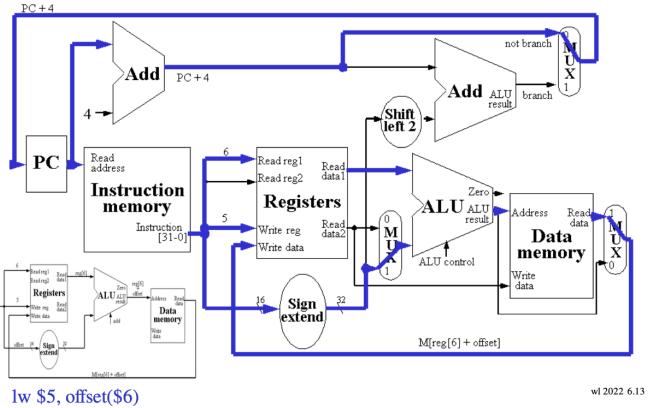
wl 2022 5.20

Single Cycle

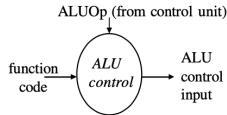
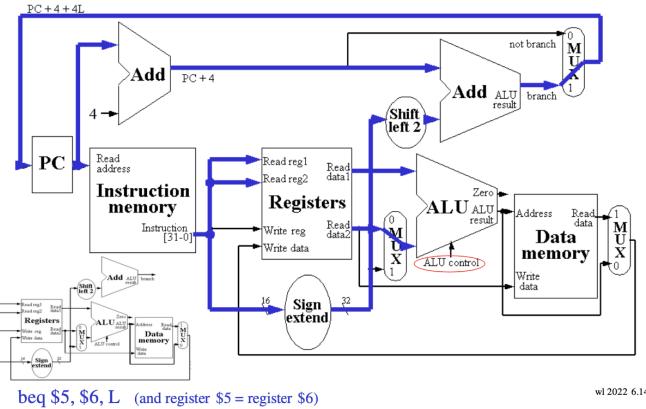
Combined datapath for R-type



Combined datapath for load



Combined datapath for branch

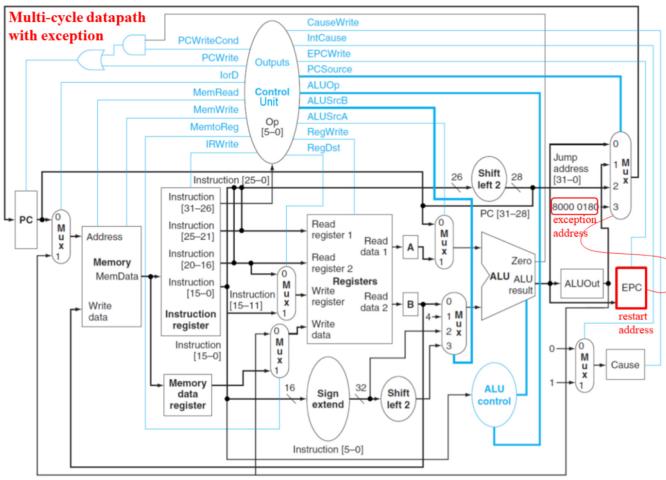


Logic for ALU control

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set-on-less-than	101010	set-on-less-than	111

from opcode input constant lecture on ALU
specify instruction type ... 2 6.15

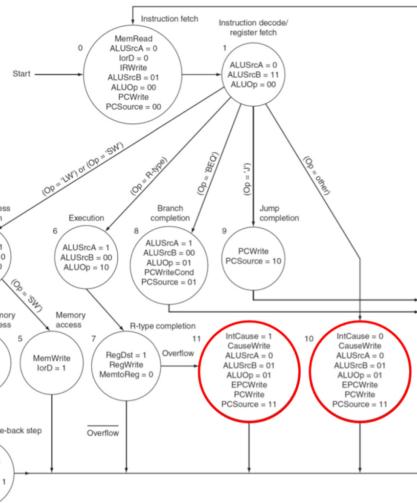
Multi Cycle with Exception



Execution steps: program format

- $IR = M[PC]$, $PC = PC + 4$; IR, PC assignment in parallel
- $A = \text{Reg}[IR_{25-21}]$, $B = \text{Reg}[IR_{20-16}]$, $\text{ALUOut} = PC + \text{sign-ext}(IR_{15-0}) \ll 2$; input to ALU store branch target
- if opcode == R-type $\text{ALUOut} = A \text{ op } B$; $\text{Reg}[IR_{15-11}] = \text{ALUOut}$ goto beginning
- if opcode == load or store $\text{ALUOut} = A + \text{sign-ext}(IR_{15-0})$;
 - if opcode == load $MDR = M[\text{ALUOut}]$; $\text{Reg}[IR_{20-16}] = MDR$
 - if opcode == store $M[\text{ALUOut}] = B$
- if opcode == beq
if ($A == B$) then $PC = \text{ALUOut}$

wl 2022 7.23



wl 2022 9.9

Microprogram: table form

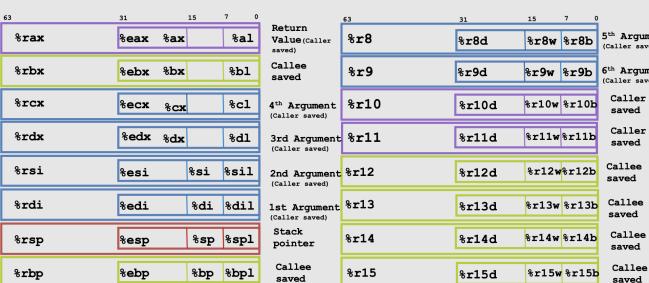
State	Label	ALU Control = FnCode		Memory	Reg Control	PC write Control	Sequencing
		SRC 1	SRC 2				
0	Fetch	Add	PC	4	ReadPC	ALU	Seq
1		Add	PC	ExtShft		Read	Dispatch 1
2	Mem1	Add	A	Extend			Dispatch 2
3	LW2				ReadALU		Seq
4					WriteMDR		Fetch
5	SW2				WriteALU		Fetch
6	Rformat1	Fn.code	A	B			Seq
7					WriteALU		Fetch
8	BEQ1	sub	A	B		ALUoutcond	Fetch
9	JUMP1					Jump addr	Fetch

wl 2022 8.13

Registers

Register	Accumulator		Counter		Data		Base		Stack Pointer		Stack Base Pointer		Source		Destination	
64-bit	RAX		RCX		RDX		RBX		RSP		RBP		RSI		RDI	
32-bit	EAX		ECX		EDX		EBX		ESP		EBP		ESI		EDI	
16-bit	AX		CX		DX		BX		SP		BP		SI		DI	
8-bit	AH AL		CH CL		DH DL		BH BL		SPL		BPL		SIL		DIL	

x86-64 Integer Registers



(Spring '21)

Introduction to Computer Architecture

Passing data - x86-64 integer registers: conventions

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Caller saved
%rbp	Callee saved	%r15	Callee saved

(Spring '22)

COMP40005 Introduction to Computer Architecture

17

Arithmetic Operations

Of interest: movz and movs

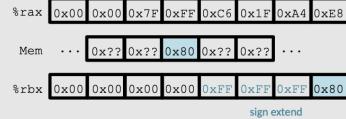
`movz __src, regDest` Move with zero extension
`movs __src, regDest` Move with sign extension

- Copy from a smaller source value to a larger destination
- Source can be memory or register; Destination must be a register
- Fill remaining bits of dest with zero(`movz`) or sign bit (`movs`)

`movz SD / movs SD`

S - size of source (b=1 byte, w=2)
D - size of dest (w=2 bytes, l=4, q=8)

Example: `movsb1 (%rax), %ebx`



(Spring'22)

Introduction to Computer Architecture

Special Arithmetic Operations

- These operations provide 128-bits

Instruction	Operation	Notes
<code>imulq src</code>	$R[\%rdx];R[\%rax] \leftarrow \text{src} \times R[\%rax]$	Signed multiplication
<code>mulq src</code>	$R[\%rdx];R[\%rax] \leftarrow \text{src} \times R[\%rax]$	Unsigned multiplication
<code>idivq src</code>	$R[\%rdx] \leftarrow R[\%rdx];R[\%rax] \bmod \text{src}; R[\%rax] \leftarrow R[\%rdx];R[\%rax] \div \text{src}$	Signed divide
<code>divq src</code>	$R[\%rdx] \leftarrow R[\%rdx];R[\%rax] \bmod \text{src}; R[\%rax] \leftarrow R[\%rdx];R[\%rax] \div \text{src}$	Unsigned divide
<code>cqto</code>	$R[\%rdx];R[\%rax] \leftarrow \text{signExtend}(R[\%rax])$	Convert to octal word

5 (Spring'22)

Introduction to Computer Architecture

13

Condition Codes

Condition codes (implicit setting)

Single bit registers

`CF` - Carry Flag (for unsigned)
`SF` - Sign Flag (for signed)
`ZF` - Zero Flag
`OF` - Overflow Flag (for signed)

Implicitly set (think of it as a side effect) by arithmetic operations (not by lea)

Example: `addl/addq Src,Dest $\leftrightarrow t = a+b$`

- CF set** if carry out from most significant bit (unsigned overflow)
- ZF set** if $t == 0$
- SF set** if $t < 0$ (as signed)
- OF set** if two's complement (signed) overflow

$(a>0 \ \& \ b>0 \ \& \ t<0) \ \mid\mid \ (a<0 \ \&\ b<0 \ \&\ t>0)$

(Spring'22)

Introduction to Computer Architecture

Condition codes (explicit setting: compare)

Explicit setting by a compare instruction

`cmpl/cmpq Src2,Src1`

Example: `copl b, a` like computing $a-b$ without setting destination

- CF set** if carry out from most significant bit (used for unsigned comparisons)
- ZF set** if $a == b$
- SF set** if $(a-b) < 0$ (as signed)
- OF set** if two's complement (signed) overflow

$(a>0 \ \& \ b<0 \ \& \ (a-b)<0) \ \mid\mid \ (a<0 \ \& \ b>0 \ \& \ (a-b)>0)$

Condition codes (explicit setting: test)

Explicit setting by a test instruction

`testl/testq Src2,Src1`

Example: `testl b, a` like computing $a \ \& \ b$ without setting destination

- Sets condition codes based on value of `Src1 & Src2`
- Useful to have one of the operands be a mask
- ZF set** when $a \ \& \ b == 0$
- SF set** when $a \ \& \ b < 0$

testl %eax, %eax

- Sets SF and ZF, check if eax is +,0,-

(Spring'22)

Introduction to Computer Architecture

Reading Condition codes (CC)

set* instructions: set low order byte to 0 or 1 based on computation of CC.

SetX instruction	Condition	Description
<code>sete dst</code>	ZF	Equal / Zero
<code>setne dst</code>	$\sim ZF$	Not equal / Not zero
<code>sets dst</code>	SF	Negative
<code>setns dst</code>	$\sim SF$	Nonnegative
<code>setg dst</code>	$\sim (SF \wedge OF) \wedge \sim ZF$	Greater (Signed)
<code>setge dst</code>	$\sim (SF \wedge OF)$	Greater or equal (Signed)
<code>setl dst</code>	$(SF \wedge OF)$	Less (Signed)
<code>setle dst</code>	$(SF \wedge OF) \mid ZF$	Less or equal (Signed)
<code>seta dst</code>	$\sim CF \wedge \sim ZF$	Above (unsigned)
<code>setb dst</code>	CF	Below (unsigned)
<code>setbe dst</code>	CF \mid ZF	Below or equal (unsigned)

23 (Spring'22)

Introduction to Computer Architecture

24

Jumping

Instruction	Condition	Description
<code>jmp target</code>	1	Unconditional (direct jump)
<code>jmp *Operand</code>	1	Unconditional (indirect jump)
<code>je target</code>	ZF	Equal / Zero
<code>jne target</code>	$\sim ZF$	Not Equal / Not Zero
<code>js target</code>	SF	Negative
<code>jns target</code>	$\sim SF$	Nonnegative
<code>jg target</code>	$\sim (SF \wedge OF) \wedge \sim ZF$	Greater (Signed)
<code>jge target</code>	$\sim (SF \wedge OF)$	Greater or equal (Signed)
<code>jl target</code>	$(SF \wedge OF)$	Less (Signed)
<code>jle target</code>	$(SF \wedge OF) \mid ZF$	Less or equal (Signed)
<code>ja target</code>	$\sim CF \wedge \sim ZF$	Above (unsigned)
<code>jb target</code>	CF	Below (unsigned)

- j*** Instructions: Jump to different part of the code indicated by **target** argument. Conditional jump depends on **condition code registers**

(Spring'22)

Introduction to Computer Architecture

26

Cache

Cache Performance

- Two things hurt the performance of a cache:
 - Miss rate and miss penalty
- Average Memory Access Time (AMAT): average time to access memory considering both hits and misses
$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$
- 99% hit rate is twice as good as 97% hit rate!
 - Assume HT of 1 clock cycle and MP of 100 clock cycles
 - 97%: AMAT = $1 + (1 - 0.97) \times 100 = 1 + 3 = 4$ clock cycles
 - 99%: AMAT = $1 + (1 - 0.99) \times 100 = 1 + 1 = 2$ clock cycles

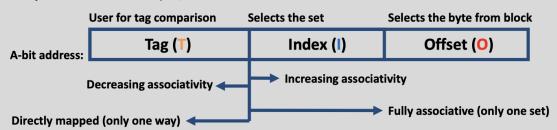
(Spring'22) COMP40005 Introduction to Computer Architecture 22 (Spring'22)

Cache Organization (3)

Cache notation:
C – size of cache
B – block size
N – associativity

Associativity (N): number of lines for each set

- Such a cache is called an “N-way set associative cache”
- We now index into cache sets, of which there are $C/B/N$
- Use lowest $\log_2(C/B/N) = l$ bits of block address
- Direct-mapped: $N=1$, so $l = \log_2(C/B)$ as we saw previously
- Fully-associative: $N=C/B$, so $l = 0$ bits



COMP40005 Introduction to Computer Architecture 23 (Spring'22)

Types of Cache Misses: 3 C's!

- Compulsory (cold) miss:
 - Occurs on first access to a block
- Conflict miss:
 - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot.
 - E.g., referencing blocks 0, 8, 0, 8, ... could miss every time
 - Direct-mapped caches have more conflict misses than N-way set-associative
- Capacity miss:
 - Occurs when the set of active cache blocks (the **working set**) is larger than the cache
 - Note: Fully-associative only has Compulsory and Capacity misses

(Spring'22) COMP40005 Introduction to Computer Architecture 18 (Spring'22)

What to do on a write hit?

- Multiple copies of data exist. What is the problem with that?
- Write-through
 - Write immediately to memory and all caches in between
 - Memory is always consistent with the cache copy
 - Slow: what if the same value (or line!) is written several times
- Write-back
 - Defer write to memory until line is evicted (replaced)
 - Need a dirty bit
 - Indicates line is different from memory
 - Higher performance (but more complex)

COMP40005 Introduction to Computer Architecture 19 (Spring'22)

What to do on a write-miss?

- Write-allocate (load into cache, update line in cache)
 - Good if more writes to the location follow
 - More complex to implement
 - May evict an existing value
 - Common with write-back caches
- No-write-allocate (writes immediately to memory)
 - Simpler to implement
 - Slower code (bad if value consistently re-read)
 - Seen with write-through caches

(Spring'22) COMP40005 Introduction to Computer Architecture 20 (Spring'22)