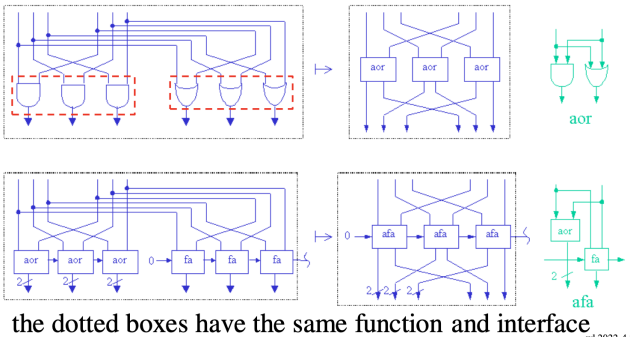
COMP40005 Introduction to Computer Architecture
Imperial College London Spring 2022

ALU

Deriving ALU cell by interleaving components

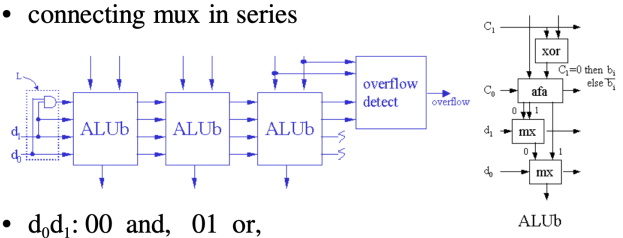
- group components together to form larger repeated unit



- the dotted boxes have the same function and interface

Selecting ALU operation

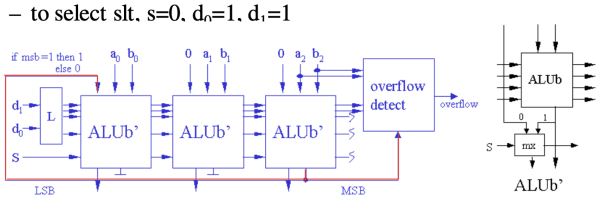
- programmable inverter for b_i (using xor)
- connecting mux in series



- d_0d_1: 00 and, 01 or,
  10 add, 11 subtract
- detecting overflow: exercise

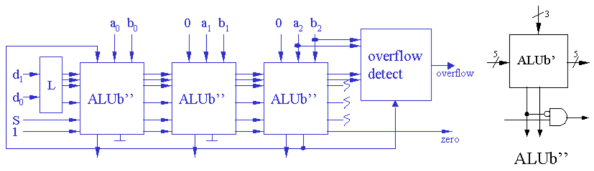
Comparison operations

- 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



Zero detection

- beq, bne: test a=b or a-b=0
- include another gate to test if output zero



- summary:

s d_0 d_1	011	100	101	110	111
function:	set on less than	and	or	add	subtract

Multiplication and Division

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

Iteration	Multi-plicand	Original algorithm Step	Original algorithm Product	Booth's algorithm Step	Booth's algorithm Product
0	0010	Initial Values	0000 0110	Initial Values	0000 0110 0
1	0010 0010	1: 0 => no operation 2: Shift right Product	0000 0110 0000 0011	1a: 00 => no operation 2: Shift right Product	0000 0110 0 0000 0011 0
2	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0010 0011 0001 0001	1c: 10 => Prod=Prod-Mcand 2: Shift right Product	1110 0011 0 1111 0001 1
3	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0011 0001 0001 1000	1d: 11 => no operation 2: Shift right Product	1111 0001 1 1111 0000 1
4	0010 0010	1: 0 => no operation 2: Shift right Product	0001 1000 0000 1100	1b: 01 => Prod=Prod+Mcand 2: Shift right Product	0001 1000 1 0000 1100 0

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

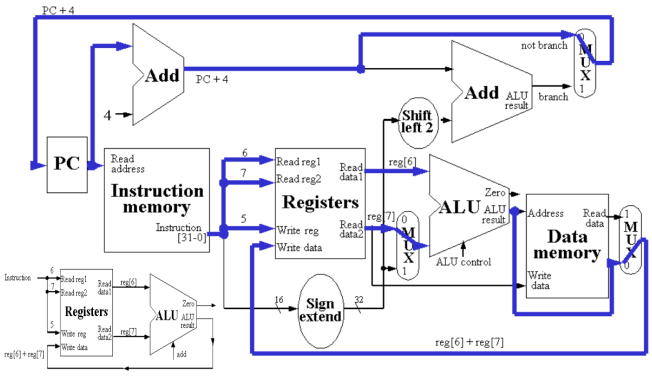
Division example using first algorithm

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 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	1: Rem = Rem - Div 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	1: Rem = Rem - Div 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0000 1000 0000 1000 0000 0100	1111 1111 0000 0111 0000 0111
4	1: Rem = Rem - Div 2a: Rem≥0 ⇒ sll Q, Q0=1 3: Shift Div right	0000 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011
5	1: Rem = Rem - Div 2a: Rem≥0 ⇒ sll Q, Q0=1 3: Shift Div right	0001 0011 0011	0000 0010 0000 0010 0000 0001	0000 0001 0000 0001 0000 0001

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

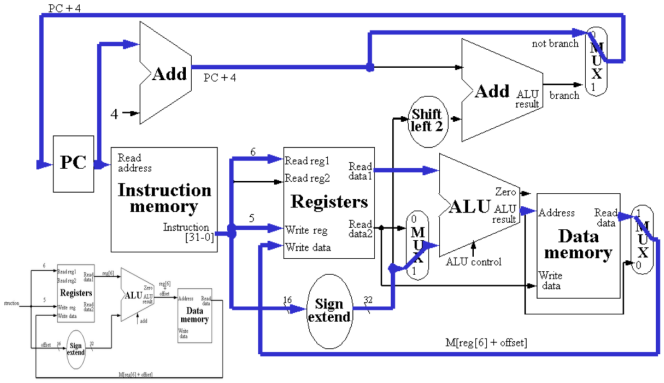
Single Cycle

Combined datapath for R-type



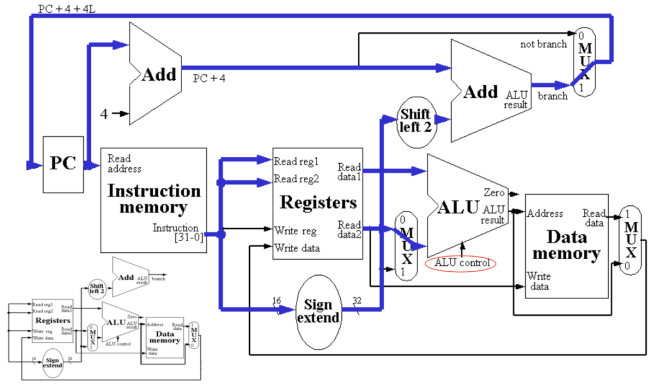
add $5, $6, $7

Combined datapath for load

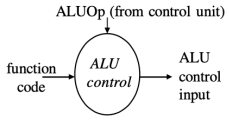


lw $5, offset($6)

Combined datapath for branch



beq $5, $6, L (and register $5 = register $6)



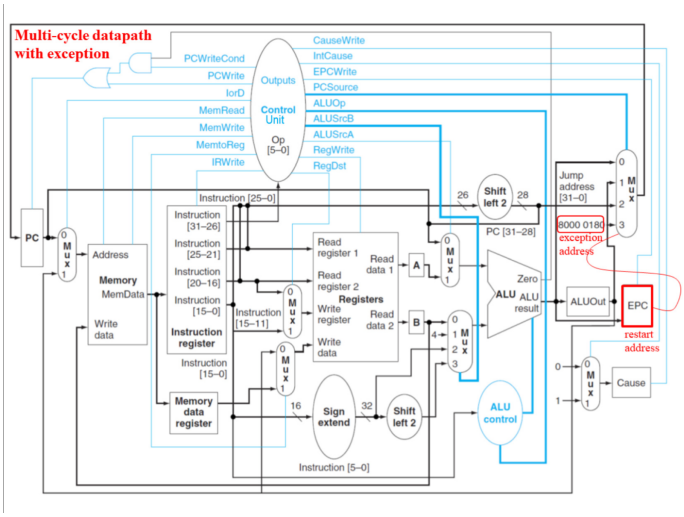
Logic for ALU control

- from opcode and fn code, derive truth table 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 specify instruction type — input — constant — lecture on ALU — output

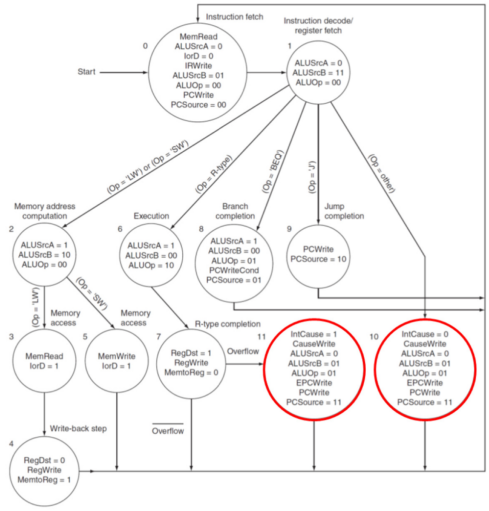
Multi Cycle with Exception



Execution steps: program format

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

wl 2022 7.23



wl 2022 9.9

Microprogram: table form

ALU Control = FnCode

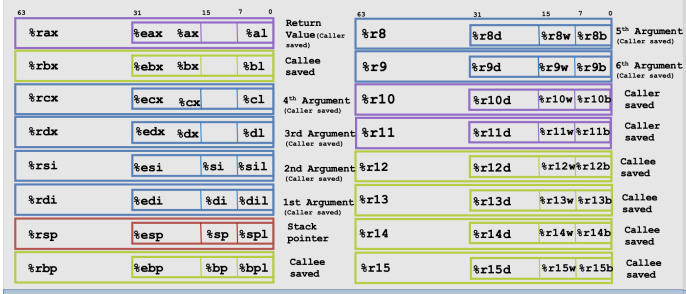
State	Label	ALU Control	SRC 1	SRC 2	Memory	Reg. Control	PC write Control	Sequencing
0	Fetch	Add	PC	4	ReadPC		ALU	Seq
1		Add	PC	ExtShft		Read		Dispatch 1
2	Mem1	Add	A	Extend				Dispatch 2
3	LW2				ReadALU			Seq
4						WriteMDR		Fetch
5	SW2				WriteALU			Fetch
6	Rformat1	Fn.code	A	B				Seq
7						WriteALU		Fetch
8	BEQ1	subt	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 35

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	Callee saved
%rbp	Callee saved	%r15	Callee saved

(Spring'22) COMP40005 Introduction to Computer Architecture 17

Arithmetic Operations and Condition Codes

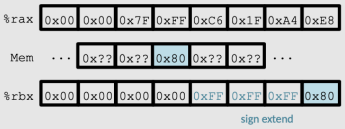
Of interest: `movz` and `movs`

`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`)

`movzSD` / `movsSD`
S – size of source (`b`=1 byte, `w`=2)
D – size of dest (`w`=2 bytes, `l`=4, `q`=8)

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



Example x86 Arithmetic Operations

- Two-operand instructions (longword variants)
- Watch out for argument order!

Instruction		Operation	Notes
`addl`	*src,dest*	dest = dest + src	Addition
`subl`	*src,dest*	dest = dest - src	Subtraction
`imull`	*src,dest*	dest = dest * src	Multiplication
`sall`	*src,dest*	dest = dest << src	Shift arithmetic left
`sarl`	*src,dest*	dest = dest >> src	Shift arithmetic right
`xorl`	*src,dest*	dest = dest ^ src	Bitwise xor
`andl`	*src,dest*	dest = dest & src	Bitwise and
`orl`	*src,dest*	dest = dest | src	Bitwise or

Quick way to multiply and divide by powers of 2

Special Arithmetic Operations

- These operations provide 128-bits

Instruction	Operation	Notes
`imulq` *src*	R[%rdx]:R[%rax] ← src X R[%rax]	Signed multiplication
`mulq` *src*	R[%rdx]:R[%rax] ← src X R[%rax]	Unsigned multiplication
`idivq` *src*	R[%rdx]← R[%rdx]:R[%rax] mod src; R[%rax] ← R[%rdx]:R[%rax] ÷ src	Signed divide
`divq` *src*	R[%rdx]← R[%rdx]:R[%rax] mod src; R[%rax] ← R[%rdx]:R[%rax] ÷ src	Unsigned divide
`cqto`	R[%rdx]:R[%rax] ← signExtend(R[%rax])	Convert to octal word

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* ↔ `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) || (a<0 && b<0 && t>=0)`

Condition codes (explicit setting: compare)

- **Explicit setting** by a `compare` **instruction**

  `cmpl/cmpq` *Src2,Src1*

  Example: `cmpl 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) || (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,-

Jumping

Instruction	Condition	Description
`jmp` *target*	`1`	Unconditional (direct jump)
`jmp` *Operand	`1`	Unconditional (indirect jump)
`je` *target*	`ZF`	Equal / Zero
`jne` *target*	`~ZF`	Not Equal / Not Zero
`js` *target*	`SF`	Negative
`jns` *target*	`~SF`	Nonnegative
`jg` *target*	`~(SF^OF)&~ZF`	Greater (signed)
`jge` *target*	`~(SF^OF)`	Greater or equal (signed)
`jl` *target*	`(SF^OF)`	Less (signed)
`jle` *target*	`(SF^OF)|ZF`	Less or equal (signed)
`ja` *target*	`~CF&~ZF`	Above (unsigned)
`jb` *target*	`CF`	Below (unsigned)

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

Reading Condition codes (CC)

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

SetX instruction	Condition	Description
`sete` *dst*	ZF	Equal / Zero
`setne` *dst*	~ZF	Not equal / Not zero
`sets` *dst*	SF	Negative
`setns` *dst*	~SF	Nonnegative
`setg` *dst*	~(SF^OF)&~ZF	Greater (Signed)
`setge` *dst*	~(SF^OF)	Greater or equal (Signed)
`setl` *dst*	(SF^OF)	Less (Signed)
`setle` *dst*	(SF^OF)|ZF	Less or equal (Signed)
`seta` *dst*	~CF&~ZF	Above (unsigned)
`setb` *dst*	CF	Below (unsigned)
`setbe` *dst*	CF | ZF	Below or equal (unsigned)

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

AMAT = Hit time + Miss rate × 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

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 $\log_2(C/B/N)$ = I bits of block address
  - Direct-mapped: N=1, so I = $\log_2(C/B)$ as we saw previously
  - Fully-associative: N=C/B, so I = 0 bits

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

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)

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