CS 321

硬件

Rapid Performance increase in computing hardware

Moore's law

Not: Clock frequency doubles every 1.5 years

Not: Clock frequency doubles every 2 years

Not: Clock frequence grows exponentially

Transistor density increases exponentially

Item which affect program performance:

choice of algorithm(s)

Language

Compiler

Architecture (ISA-Instruction Set Architecture (analogous to API))

- ADD x0, x1, x2

Architecture (implementation) and memory system

I/O system.

Computer Organi and design

hardware software interface

arm addition

How did computers become so fast?

Design for the future

Use abstraction effectively

Design for the common case

Exploit parallelism

Predict spatial locality - Prefetch data the we expect to need soon based on locations that we have use recently

Temporal locality - revising locations

Use hierarchical structures

Main memory - 10^5-10^6 cycles to get data from main memory

Register - 0 cycles

Be dependable L1 cache - ~10 cycles

L2 cache - ~100 cycles

L3 cache - ~1000

Five classic components of a computer

- Input - keyboard, network, serial port, hard drive, etc.
- Output - monitor, printer, serial port network, hard drives, etc.
- Memory - RAM, volatile storage
- Datapath - Pathways that data take to and from input, memory, and the CPU
- Control - How the computer "decides" what to do next: what data do we need, what instruction do we excute on the data, what do we do with the result

Measuring performance

CPU Time = CPU clock cycles x clock cycle time

Clock cycle time = 1 / clock rate

CPU Time = CPU clock cycles / clock rate

To improve program performance, either reduce clock cycles, or increase clock rate (reduce cycle time)

$$\text{CPU Time} = \frac{\text{Instruction count} \times \text{Cycles per instruction}}{\text{clock rate}}$$

for reduce the voltage to the processor, then it takes longer for transistors

Voltage is related to stuff

When flow is low, your voltage takes longer to reach steady state. And that means you have to reduce your switching speeds

A program runs in 15s on a 2.5 GHz processor (CPUa). We want to design a computer (CPUb) that can run it in 10s.

15 * 2.5e9 = # of cycle = 3.75e10

10s = 3.75e10 * clock cycle time

Frequency (clock rate) 3.75e9 → 3.75 GHz

15/10 = x/2.5

How long do they take to run 3.5 GHz

A	B	C	
4	3	6	
	cycles		
450	375	200	Program 1
375	130	250	Program 2

450×4+375×3+200×6 = 4125 cycles

375×4+130×3+250×6 = 3390 cycles Faster

4125 / 3500000000 = 0.0000011785714 = 1.1e-6s

3390 = 9.7e-7s

one more equation, boiling everything down to its simplest form:

$$\text{Time} = \frac{\text{Seconds}}{\text{program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{clock cycles}}{\text{Instructions}} \times \frac{\text{Second}}{\text{clock cycle}}$$

The power wall

Begining in the early to mid oughts, processors started to become difficult to cool

Dynamic power

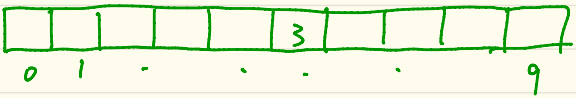
Energy is proportional to capacitative load * voltage squared

Energy is power * time

Power is proportional to voltage squared * frequency

usually 4 bytes

int a[10];



a[5] = 3

a[5] = 3

5·4=20 bytes

(ampersand:)

at 5

this is an address (location)

at5 == &a[5]

exactly the same

a[5] == *(at5)

int a[10]

char *p = a

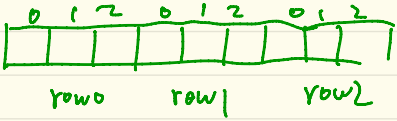
int m[3][3];

rows

	0	1	2
0			
1			7
2			

m[1][2]=7

a



int a[m][n];

a[p][q]=row;

&a[p][q] = a+(p*n+q)*sizeof(int)

c

$00000000_2 = 0_{10}$

$00000001_2 = 1_{10}$

$+\ 11111110_2 = -1_{10}$

$11111111_2 = -0_{10}$

(per second)

instruction × CPI = Clock Rate

no. instruction = clock rate × time

P1 has a 3 GHz clock rate and a CPI of 1.5

P2 25 1

4 2.2

① performance? $P_1: \frac{3\times10^9}{1.5}$ $P_2 = \frac{25\times10^9}{1}$ ✓ $P_3: \frac{4\times10^9}{2.2}$ instructions per seconds.

② excute 10 seconds no. of cycles? instruction?

cycle: $P_1: 3\times10^9\times10 = 3\times10^{10}$

$P_2: 25\times10^{10}$

$P_3: 4\times10^{10}$

③ Reduce 30% time, and increase 20% CPI, New clock rate?

$0.7 \times \text{Exaction Time} = \frac{\text{no. instructions} \times \text{CPI} \times 1.2}{\text{New Clock Rate}}$ $\text{Time} = \frac{\text{no. instruction} \times \text{CPI}}{\text{clock Rate}}$

Week 3

suffixes means there's an immediate in the immediate is a hard coded constant number

```
main:
    ADDI X10, X2R, #512
    ADD  X11, X2R, X2R        no immediate
                              adding to register

    start_init:
subtract
        SUB X12, X10, X2R
        CBZ X12, end_init
        STUR X10, [X11, #0]
        SUBI X10, X10, #1
        ADDI X11, X11, #8
        B start_init

    end_init:
        ADD X0, X2R, X2R
        PRNT X0
        ADDI X1, X2R, #512
        BL selection_sort
        B done

    swap:
        LDUR X9, [X0, #0]     // load *a
        LDUR X10, [X1, #0]    // load *b
        STUR X10, [X0, #0]    // store *b into a
        STUR X9, [X1, #0]     // store *a into b
        BR LR
```

ARMv8

LEGv8 Assembly Longwage

Find smallest:

```
ADD X9, X2R, X2R
ADDI X10, X2R, #1
```

done:

--> DUMP

Registers have names X0 – X31

Example: Compile the line of C code: y = x + a[10] + 3, where y is in X21, x is in X20 and address of a in X22

```
LDUR X21, [X22, #80]   // stores the value in a[10] in X21 (y)
                       // 80 (not 10) because each value is 8 bytes
ADDI X21, X21, #3      // X21 (y) = a[10] + 3
ADD X21, X20, X21      // X21 (y) = (a[10] + 3) + x
```

Jumping

B (branch) jumps unconditionally, takes a label

CBZ (compare and branch if zero). takes a register and a label

CBNZ (not zero). takes a register and a label

Labels are simply "name:" in the program

LDUR	Doubleword from memory to register	Load Register
STUR	Doubleword from register to memory	Store Register

```
if(x==y)
    x++;
else
    y++;
```

x -> X21

y -> X22

```
SUB X9, X21, X22        // X9 = x-y
CBNZ increment_y        // if x - y != 0, jump to increment y
ADDI X21, X21, #1       // add 1 to x
B end                   // jump to the end of program
increment_y:
ADDI X22, X22, #1       // add 1 to y
end:
```

```
while(str[i])
    i++;
```

str -> X21

i -> X22

Use LSL (logical shift left) to multiply by 8

```
LDUR X9, [
LDUR X9, [X21, X22 * 8]
```

```
LSL X10, X22, #3        // i*8
ADD X10, X10, X21       // X10=str+i*8(X10=&str[i])
LDUR X9, [X10, #0]      // fetch str[i] into X9
CBZ X9, end             // if Str[i]=0, branch to end
ADDI X22, X22, #1       // increment i
B top
end
```

Making a procedure call

In order to call a procedure, the program must:

1) place the parameters - In ARM, the programeters go in register x0 - x7
2) Transfer control - That is, jump to the procedure BL (branch and Link)
3) Allocate storage - for saved registers, "automatic variables"
4) Perform caculation
5) place the result - Return value goes into parameter registers (x0 - x7)
6) Return control - Restore saved values and jump back to the caller.

Registers and instructions to facilitate this:

registers:

- sp stack pointer, x28 — bottom
- FP frame pointer, x29 — Current func context is indexed by frame pointer and Link Register, holds the address of branch that brought us to the current procedure
- LR return address, x30

instructions:

- BL address : Branch and link (jumps to address and store PC in LR)
  PC is "program counter", special register that stores the address of the current instruction
- BR register : Branch to register (Jumps to LR + 4 (instructions are 32 bits); if no +4, the BL instruction would execute again, forever.)

```
long long fib(long long n){
    if (n < 2){ return n }
    return fib(n-1) + fib(n-2)
```

bottom of our version

```
fib:
    CBZ x0, done
    SUBI x9, x0, #1
    CBNZ x9, body
done:
    BR LR                    Branch to register    Link to register
body:
    SUBI sp, sp, #24
    STUR x9, [sp, #0]
    STUR x20, [sp, #8]
    STUR LR, [sp, #16]
    ADD x9, x0, x2R
    SUBI x0, x19, #1
    BL fib
    ADD x20, x0, x2R
    SUBI x0, x19, #2
    BL fib
    ADD x0, x20, x0
    LDUR LR, [sp, #16]
    LDUR x20, [sp, #8]
    LDUR x19, [sp, #0]
    ADDI sp, sp, #24
    BR LR
```

make room on stack and backs up saved register

set up parameter

make recursive call

save the result

second call

Add two results save in x0

restore stack

return to caller

```
long long int fact (long long int n){
    if (n<1) {
        return 1;
    }else{
        return (n * fact(n-1));
    }
}
```

```
fact:
    SUBI SP, SP, #16
    STUR LR, [SP, #8]      // LR => to other function
    STUR X0, [SP, #0]
    SUBIS 2XR, X0, #1      // test for n<1
    B.GE L1                // if n >=1, go to L1
    ADDI X1, X2R, #1       // return 1
    ADDI SP, SP, #16       // pop 2 items off stack
    BR LR                  // return to caller
L1: SUBI X0, X0, #1        // n >=1 : argument gets (n-1)
    BL fact                // call fact with (n-1)
    LDUR X0, [SP, #0]      return from BL: restore argument n
    LDUR LR, [SP, #8]      restore the return address
    ADDI SP, SP, #16       adjust stack pointer to pop 2 items
    MUL X1, X0, X1         // return n * fact(n-1)
    BR LR                  // return to caller
```

10	11	12	13	14	15
a	b	c	d	e	f
1010	1011	1100	1101	1110	1111

Compare:
```
SUBIS 2XR, X0, #1
B.GE L1          // n >=1 , go L1
SUBS
```

```
long long int leaf_example(long long int g, ....h, .... i, .... j){
        long long int f
        f=(g+h)-(i+j);
        return f;
}
```

```
SUBI SP,SP,#24
STUR x9,[SP,#0]
STUR x10,[SP,#8]
STUR x11,[SP,#16]
ADD x9,x0,x1
ADD x10,x2,x3
SUB x11,x9,x10
ADD x0,x2R,x11
LDUR x11,[SP,#16]
ADDI SP,SP,#24
```

```
leaf_example:
    SUBI SP, SP, #24
    STUR x10,[SP,#16]
    STUR x9,[SP #8]
    STUR x19,[SP,#0]
    ADD x9, x0, x1
    ADD x10, x2, x3
    SUB x19, x9, x10
    ADD x0, x19, ZXR
    LDUR x19,[SP,#0]
    LDUR x9,[SP,#8]
    LDUR x10,[SP,#16]
    ADDI SP, SP, #24
    BR LR
```

```
void strcpy(char x[], char y[]){
    size_t i;
    i=0
    while((x[i]=y[i]) != '\0')
        i=i+1;
}
```

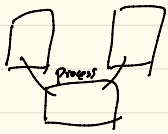
```
f g h i j        A B
x0 1 2 3 4       x6 x7
B[8] = A[i-j]
    SUB x9, x3, x4
    LSL x10, x9, #3     [i-j]
    ADD x10, x10, x6    A[i-j]
    LDUR x11, [x10, #0]
    STUR x11, [x7, #64]
```

LDXR — load exclusive register

STXR — store exclusive register

```
LDXR  Xn, [Xm, #0]
  ⋮
STXR  Xp, Xq, [Xm]
```

Xq signals success (0) or failure (1) of the store



Atomic Exchange

```
again:
    LDXR  x10, [x20, #0]
    STXR  x23, x9, [x20]
    CBNZ  x9, again
    ADD   x23, x10, #0
```

Lock:

```
        ADDI  x11, x2R, #1
again:  LDXR  x10, [x20, #0]
        CBNZ  x10, again
        STXR  x11, x9, [x20]
        CBNZ  x9, again
whendone: STUR  x2R, [x20, #0]
```

if x10 has the base of array A and x21 corresponds to h, A[30] = h + A[30] + 1

```
LDUR  x9, [x10, #240]
ADD   x9, x21, x9
ADDI  x9, x9, #1
STUR  x9, [x10, #240]
```

2+4+8

4 +8+16
1 2

ADD R

SUBS R
x11 x15, x14

2^5 2^4 1 2 4 8

8 16 1 2 8

encoding instruction

ADD x20, x13, x27 R $Rd = Rn \times Rm$

opcode	Rm	shamt	Rn	Rd
11	5	6	5	5
10001011000	11011	000000	01101	10100

no space

ADDI x7, x12, #73 I $Rd = Rn + Imm$

opcode	Imm	Rn	Rd
10	12	5	5
1001000100	000001001001	01100	00111

Build a fence 100ft long

posts go every 10 ft

How many posts do you need?

1 2 3 4 5 6 7 8 9 10

R	opcode	Rm	shamt	Rn	Rd
	11 bits	5	6	5	5
I	opcode	Imm	Rn	Rd
	10	12	5	5
D	opcode	address	op2	Rn	Rt
	11	9	2	5	5
B	opcode	BR_addr	
	6	26	
CB	opcode	COND BR_addr	Rt
	8	19	5
IW	opcode	imm	Rd
	11	16	5

opcode	Rm	shamt	Rn	Rd
1624	9	0	10	11

=> SUB x11, x10, x9

AB x6 x7
fghij 0 1 2 3 4

B[8] = A[i-j) fghij x0,1,2,3,4 AB address x6, x7

```
SUB x9, x3, x4      // x9 = i-j
LSL x9, x9, #3
ADD x9, x9, x6
LDUR x10, [x9, #0]
STUR x10, [x7, #64]
```

B[8] = A[i] + A[i]

```
LSL x9, x3, #3      [i]
ADD x9, x9, x6      A[i]
LDUR x10, [x9, #0]  拿一個變數 x7.
LSL x11, x4, #3
ADD x11, x11, x6
LDUR x12, [x11, #0]
ADD x12, x12, x10
STUR x12, [x7, #64]
```

32+16

LDUR X15, [X6, #48] LDUR : D R[Rt] = M[R[Rn] + DTAddr]

opcode	Addr	op	Rn	Rt	R-type
11	9	2	5	5	
11111000010	000110000	00	00110	01111	

B foo == B -5(not -20)

foo:

Inst ← instruction 0 — 0 bytes offset

Inst ← 1 — 4 bytes

Inst ← 2 — 8 bytes

Inst ← ⋮ 12, 16, 20

opcode	BR addr	000101
6	26	111010
000101	1……1011 (23)	1……111011

B foo

0 → 0000

4 → 0100

8 → 1000

12 → 1100

⋮

CB3 X11, foo (+4)

CB type instruction

opcode	offset	Rt
8	19	5
10110100	00……0100 (16)	01011

CB2

Inst

Inst

Inst

foo:

Inst

movk X10, #96, LSL 5L

movz

64bit register | unused | 96 | unused | 0 | - IW type

63 48 47 32 31 16 15 0

opcode	imm	Rt
11	16	5
11110010	1000000001100000	01010
op	imm	register

shift

0	00
16	01
32	10
48	11

1111 1000 0000 0100 0000 0010 0010 0001 1011100

opcode dt addr op Rn Rt

Rt = Rn + addr

This is always first. Here

STUR X1, [X17, #64]

ADD x_1, x_2, x_3

Rd M Rm

1110 1010 0010 1011 0000 0001 0100 1001

opcode Rm shamt Rn Rd

ANDS, x9, x10, x11

LSL

LSR.

1101 0011 0110 0000 0011 1101 0010 1001

LPcode Rm shamt Rn Rd

Rd = Rn << shamt

Rm not use

LSL X9, X9, #15

1101 0010 1100 1111 1111 1111 1110 1110

opcode op Imm Rd

MOVZ X14, #32767, LSL 32

00 - 0
01 - 16
10 - 32
11 - 48

16 16 16 11

Exam 2 Notes

Processors P_1, P_2, P_3 excute same instruction.

P_1 3GHz clock rate a CPI of 1.5

P_2 2.5GHz clock rate CPI 1.0

P_3 4GHz clock rate CPI 2.2

Highest performance?

$\frac{3\times10^9}{1.5} = 2\times10^9$

$\frac{2.5\times10^9}{1} = 2.5\times10^9$ P_2 ✓

$\frac{4\times10^9}{2.2} = 1.82\times10^9$

If the processors each excute a program in 10s, find the number of instructions.

$3\times10^9\times10 = 3\times10^{10}$ cycles $\frac{3\times10^{10}}{1.5} = 2\times10^{10}$ inst

$2.5\times10^9\times10 = 2.5\times10^{10}$ cycles $\frac{2.5\times10^{10}}{1} = 2.5\times10^{10}$ inst

4×10^{10} cycles, $\frac{4\times10^{10}}{2.2} = 1.82\times10^{10}$ instructions

Reducing the exection tim by 30%, but this leads to an increase of 20% in the CPI? Clock rate to get this time reduction

$0.7\times\text{Execution Time} = \frac{\text{inst}\times\text{CPI}\times1.2}{\text{New clock rate}}$

New clock Rate $= 1.71\times$ clockRate

P_1: 3×1.71

P_2: 2.5×1.71

P_3: 4×1.71

```
SUBIS ZXR, X0, #1   // test for X0 > 1
B.LT L1             // if X0>1, go L1
SUBS X11, X10, X9   // test for X10, X9
B.GT L1             // X10>X9, then L1
```

LSL for [ ]

```
LSL X11, X10, #3    // X11 = [i] f×8
ADD X11, X11, X6    // X11 = A[i] = &A[i]
LDUR X12, [X11, #0] // f = A[i] = f = A[i]
STUR X12, [X7, #64] // B[8] = A[i]
```

```
LSL X10, X22, #3
ADD X10, X10, X25
LDUR X11, [X10, #0]
SUB X12, X11, X24
CBNZ Exit
ADDI X22, X22, #1
B Loop
```

Determine if equal

SUB ⇒ CBZ // CBNZ

```
SUBIS X2R, X0, #1   // X0 ≥ 1
B.GT ----
```

Determine >, <, >=, <=

SUBS ⇒ B.GT, LT, GE, LE

Generally, at the end, could be

```
ADD X0, returnValue, ZXR
BR LR
```

Use STUR, SP, #--

When using BL ...

```
SUBI SP, SP, #
STUR X10, [SP, #0]
⋮
LDUR X10, [SP, #0]
ADDI SP, SP, #
BR LR
```

if want to return X11

```
ADD, X0, X2R, X11
BL RL
```

R	opcode	Rm	shamt	Rn	Rd
	11 bits	5	6	5	5
I	opcode	Imm	Rn	Rd	
	10	12	5	5	
D	opcode	address	op2	Rn	Rt
	11	9	2	5	5
B	opcode	BR_addr			
	6	26			
CB	opcode	COND	BR_addr	Rt	
	8	19		5	
IW	opcode	imm	Rd		
	11	16	5		

```
SUBI SP, SP, #24
STUR X10, [SP, #0]
STUR X11, [SP, #8]
STUR X12, [SP, #16]
ADD X10, X0, X1
ADD X11, X2, X3
SUB X12, X10, X11
ADD X0, X12, X2R
LDUR
⋮
ADDI SP, SP, #24
BR LR
```

· Measuring performance

CPU Time = CPU clock cycles × clock cycle time

Clock cycle time = 1 / Clock rate

CPU Time = CPU clock cycles / clock rate

$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{Cycle per instruction}}{\text{Clock rate}}$

$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} = \frac{\text{clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{clock cycle}}$

Chapter 3

3.3 Multiplication

$1 \times 0.15 \quad 3 \times 0.15 \quad 2 \times 0.4 \quad 3 \times 0.3$

$0.15 + 0.45 + 0.8 + 0.9$

15

0.6

1.7

$\frac{2.3 \times 10^6}{2.8 \times 10^9}$

$\frac{0.2 \times 10^{-4} \times 2.8 \times 10^9}{10^6}$

2.8×10^6

0.82×10^{-3}

1

$\frac{5 \times 10^{-4} \cdot 3.1 \times 10^9}{10^6}$

1.55×10^6

2 0.15 1 0.15 2) 0.3

0.4

0.3 0.15 0.8 0.3

0.6 1.4

0.15

$\frac{1.55 \times 10^6}{3.1 \times 10^9}$

5×10^{-4}

1233

1252

13	00000 1101
× 21	× 0000 1010 1
13	----- 1101
26	---- 1101
273	+ ---- 1101
	1 0001 0001

$256 + 16 + 1 = 273$

sheaffer@coms327-vm:

~/legv8emul$ ./legu8emul /home/sheaffer/legu8emul assignment1.legv8asm

1323

0.15 0.15 0.4 0.3

$0.15 \times 10^5 \quad 0.45 \times 10^5 \quad 0.8 \times 10^5$

0.9×10^5

2 1 2 1

0.3 0.15 0.8 0.3

return i:

ADD X0, X9, XZR

BR LR

```
ADDI X9, ZXR, #0      // i=0
- - - -                i < final
    SUBS X15, X9, X2   // i - final
    B.GT return
    LSL X14, X9, #3    // X14 = i×8
    ADD X10, X0, X9
    LDUR X11, [X0, #0] // X11 = addr[i]
    SUBS X13, X11, X1  // addr[i] - val
    B.GE return        // return i
    ADDI X9, X9, #1    // i++
    B - - - -
```

return Pos in Sorted

(addr, number)

i < number

i++

addr[i] =

4, 2, 6, 7, 0

i	
25	15
40	30
10	10
5	5
5	5
5	0
10	5

70

Float Point

Defines formats to represent floating point numbers

32-bit IEE 754

1 bit sign (MSB), followed by 8 bits of exponent, and 23 bits mantissa

Sign is $-1 \times$ sign bit followed by 1.mantissa $\times$ 2^(exponent -127)

Ex 1:

5 base 10 = 101 base 2 To convert to IEEE 754, first normalize

101 → 1.01 move 2

1.01×2^2

To calculat biased exponent

$x - 127 = 2$

$x == 129 \rightarrow 10000001$

Sign Exponent Mantissa

0 10000001 01000000000000000000000

Ex2: 2 base 10 = 10 base 2

10 → 1.0 move 1

Normalize → 1.0×2^1

$x - 127 = 1 \rightarrow x = 128$

Sign Exponent Mantissa

0 10000000 00.....0 (23)

7 base 10 = 111 base 2

1.11×2^3

$x - 127 = 2 \quad x = 129$

0 10000001 1100....

8+24

9: 1001

1.001×2^3

$x - 127 = 3 \quad x = 130$

0 10000010 00100..

Ex4: -11.84375 : -1011.11011

0.84375 = 1

0.4375 = 1

0.375 = 0

0.75 = 1

0.5 = 1

x = 130

10000010 01111011000

3.75 = 11.11 base 2

1.111×2^1

$x - 127 = 1 \quad x = 128$

0 10000000 1110000

float number

: 3.75

0.75 × 2 = 1.5 1

0.5 × 2 = 1 1

0.75 ⇒ 11

3 ⇒ 11

⇒ 11.11

Sep 21

To add floating point numbers:

* shift the significand of the number 20% the smaller to the right.
* add the significands
* normalize
* round.

$$\begin{array}{r} 0.427 \times 10^4 \\ 4.27 \times 10^3 \\ + \ 5.13 \times 10^4 \\ \hline 0.5557 \times 10^4 \end{array}$$

$5_{10} = 1.01 \times 2^2$

$3.75_{10} = 1.111 \times 2^1$

$$\begin{array}{r} 1.01 \times 2^2 \\ 0.1111 \times 2^2 \\ \hline 10.0011 \times 2^2 \\ 1.00011 \times 2^3 \end{array}$$

$2_{10} = 1.0 \times 2^1$

$11.84375_{10} = 1.01111011 \times 2^3$

$$\begin{array}{r} 0.01 \times 2^3 \\ 1.01111011 \times 2^3 \\ \hline 1.101111011 \times 2^3 \end{array}$$

13.84375

To multiply two floating point numbers:

* Add the exponents
* multi the significands
* normalize
* round
* sign

$5_{10} = 1.01 \times 2^2$

$2_{10} = 1.0 \times 2^1$

$$\begin{array}{r} 1.01 \\ 1.0 \\ \hline 101 \times 2^3 \end{array}$$

$3.75_{10} = 1.111 \times 2^1$

$5_{10} = 1.01 \times 2^2$

$$\begin{array}{r} 1.111 \\ \times \ 1.01 \\ \hline 1111 \\ 1111 \\ \hline 10.0101 \times 2^3 \\ 1.00101 \times 2^4 \end{array}$$

18.75

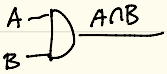
Sep 23

Digital Logic

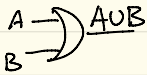
combinational logic - logic whose value (result) depends only on its inputs.

seq logic - logic whose value depends on its inputs and some combinational

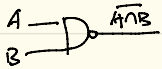
AND Gate



OR Gate

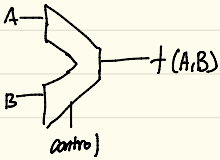


NAND Gate

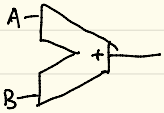


Not Gate

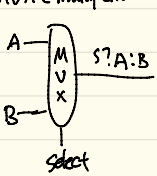
ALU Arithmetic Logic Unit

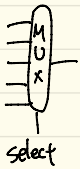


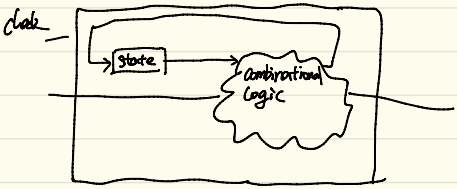
fixed Function

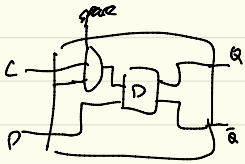


MUX (multiplexer



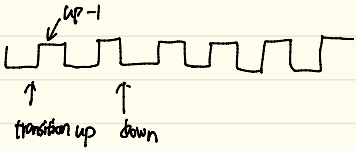


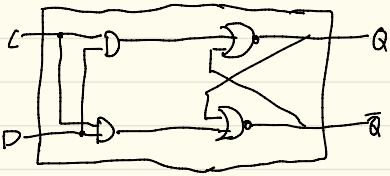




D-flipflop

A D-flipflop stores a single bit, the value of D when the clock "goes up". Q is the previous stored value

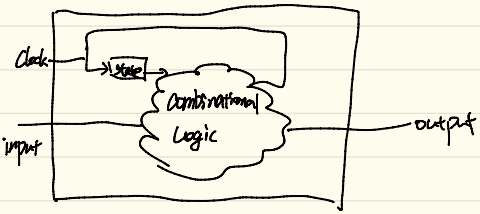




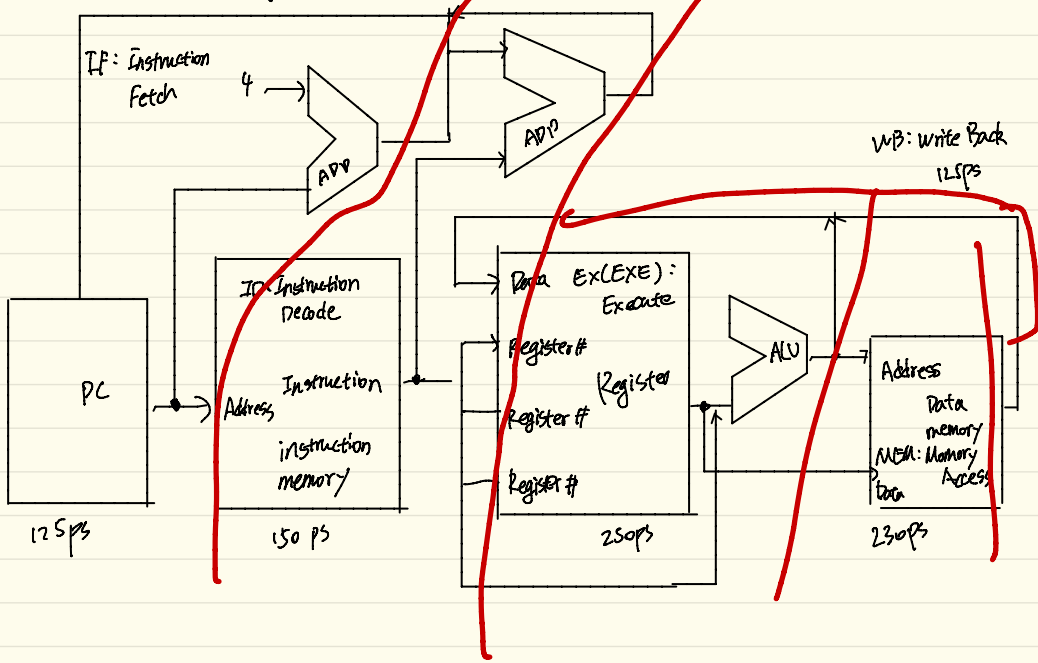
C	D	Q
0	0	0
0	1	1
1	0	0
1	1	1

Sep 25

sequential logical



Sep 28



It takes 1 ns to reach steady state → 1 GHz clock frequency

IF: Instruction Fetch

ID: Instruction Decode

EX: Execute

MEM: Memory Access

WB: Write Back

	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2		IF	ID	EX	MEM	WB				

Doesn't work
Because Inst. 2 is using x9 in cycle 4 before it is produced in cycle 5!

	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2		IF	ID	-	-	EX	MEM	WB		

Isn't avaiable until after inst. 1 complete WB
so, we need to stall inst 2's EX stage until after that time.

	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2		IF	ID	EX	MEM	WB				

We can't implement some control logic and data pathways to get the result of inst 1's add into inst 2's execute stage as soon as it is avaiable
This is called data forwarding

LDUR x10, [x10, #0]
ADD x9, x10, x11
Stall two cycles while waiting for value from memory to be written to reg. file

1	2	3	4	5	6	7	8	9	10
IF	ID	EX	MEM	WB					
	IF	ID	-	-	EX	MEM	WB		

with data forwarding

1	2	3	4	5	6	7
IF	ID	EX	MEM	WB		
	IF	ID	—	EX	MEM	WB

pipeline diagram

Sep 30

x = y + z	x	y	z	v	w
w = z + v	#0	#8	#16	#24	#32
	x0	x1	x2	x3	x4

```
LDUR x1, [sp, #8]
LDUR x2, [sp, #16]
ADD  x0, x1, x2
LDUR x3, [sp, #24]
ADD  x4, x2, x3
```

7. STUR X4, [SP, #32]

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	IF	ID	EX	MEM	WB										
2		IF	ID	EX	MEM	WB									
3			IF	ID	–	EX	MEM	WB							
4				IF	ID	–	EX	MEM	WB						
5					IF	ID	–	EX	MEM	WB					
6						IF	ID	–	–	EX	MEM	WB			
7							IF	ID	–	–	EX	MEM	WB		

1	IF	ID	EX	MEM	WB					
2		IF	ID	EX	MEM	WB				
5			IF	ID	EX	MEM	WB			Move third load up to fill bubble
3				IF	ID	EX	MEM	WB		ADD X1 and X2, X2 is forward
4					IF	ID	EX	MEM	WB	Forwarding value from mem
6						IF	ID	EX	MEM	WB Both values in reg. file
7							IF	ID	EX	MEM WB Forwarding from mem.

1. LDUR X1, [SP, #8]
2. LDUR X2, [SP, #16]
3. ADD X0, X1, X2
4. STUR X0, [SP, #0]
5. LDUR X3, [SP, #24]
6. ADD X4, X2, X3
7. STUR X4, [SP, #32]

Compilers (actually optimizing assemblers) do this automatically. Most modern processors also support out-of-order issue to do this dynamically, on the fly.

blue line : control signal

Branch Prediction

```
SUBI x0,x0,#1
CBZ x0,foo
ADD ...
foo:
SUB ...
```

without branch prediction

	1	2	3	4	5	6	7	8	9	10	11	12
SUBI	IF	ID	EX	MEM	WB							
CBZ		IF	ID	EX	MEM	WB						
???			-	-	IF	ID	EX	MEM	WB			

Static Branch Prediction

"static" always means "known at compile time".

Forward branches not taken/backward branches taken

```
while (foo) {
  stuff...
}
other stuff.
```

with forward taken/backward taken

	1	2	3	4	5	6	7	8	9	10	11	12
SUBI	IF	ID	EX	MEM	WB							
CBZ		IF	ID	EX	MEM	WB						
ADD			IF	ID	EX	MEM	WB					
SUB				IF	ID	EX	MEM	WB				

All branches taken

	1	2	3	4	5	6	7	8	9
SUBI	IF	ID	EX	MEM	WB				
CBZ		IF	ID	EX	MEM	WB			
SUB			IF	ID	EX	MEM	WB		

All branches not taken

	1	2	3	4	5	6	7	8	9
SUBI	IF	ID	EX	MEM	WB				
CBZ		IF	ID	EX	MEM	WB			
ADD			IF	ID	EX	MEM	WB		
SUB				IF	ID	EX	MEM	WB	

Dynamic Branch Prediction

Use a 1-bit buffer, called a "Branch Prediction Buffer"

Keep a table of BPBs index by the low-order bits of the PC

Works because of the principal locality — that is, instructions that executed recently are likely to be executed again soon. branches that want a particular direction are likely to go that direction again soon,

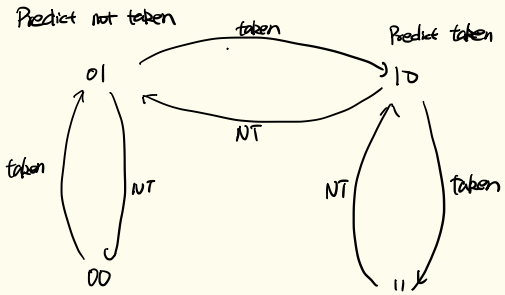
Each bit maps to a branch, and a value of 0 in the buffer means we predict not taken, while 1 means predict taken.

```
for(i from 0 to n)
    do stuff.
do other buffer
```

2 bit branch prediction buffer

Add a second bit to the BPB, highest-order gives the prediction, lowest-order bit provides hysterisis, this reduce overcorrection. These bits are a saturating counter (adding 1 to 11 is 11; subtracting 1 from 00 is 00)

```
for( i from 0 to n)
  for( j from 0 to m) // BPB: 0|
      do stuff
  do other stuff.
```



Predicting Branch Targets

B too

...

too :

ADD

Need to insert two cycles of bubbles to resolve Branch destination

	1	2	3	4	5	6	7	8	9	10	11	12
B	IF	ID	EX	MEM	WB							
ADD		–	–	IF	ID	EX	MEM	WB				

Use a Branch Target Buffer (BTB) : A cache that is indexed by a branch instruction address and stores the address of the branch target.

If we correctly predict our branch target.
we can eliminate these bubbles :

	1	2	3	4	5	6
B	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

What happens in the event of a branch misprediction? Whether the branch (taken/not taken) is mispredicted or the branch target is mispredicted, the result is the same: we are executing code from an incorrect address and it is handled in exactly the same way. The speculatively issued instructions that were erroneously issued must be invalidated by de-asserting reg-write and mem-write. This effectively converts the instructions to no-ops. The PC is the updated to the correct value and execution continues.

Exceptions

Exceptions include: Reset
I/O
O/S Invocation: O/S and HW have to work together
Floating point errors
Undefined instructions
Mal-function

When an exception occurs: We may have to terminate the process
It may be that the hardware can recover
It may be that the software can recover

All of these are essentially branches, and handling them is (almost) identical to handling control hazards

10.07

To keep track of the address of the exception:

Two registers: ELR - Exception link register
Records the address when the exceptions occured, which allows control to return here after exception is handled

ESR - Exception Syndrome Register
Used to signal the reason for the exception

Handling exceptions involves a contract between the HW and the OS

HW will: Stop the offending instructions
Complete any addr instruction
Invalidate subsequent, already issued instructions
Set ELR and ESR
Branch to a pre-arranged address - On Arm, this is 0x/c080000

SW(OS) will: Check the cause of the exception
Act appropriately

Precise vs. Imprecise Exceptions

Precise Exceptions - We know exactly where the exception occured
Imprecise Exceptions - We do not know exactly where the exception occured
Start to matter when we have: Speculative Exceptions
out-of-order Exception

Multiple Issue
Very deep Pipelines

10.09

Instruction-level Parallelism (ILP) – The inherent parallelism that exists between instructions.

Exploit it: Pipelining – Deeper pipelines allow for higher IPC, but require more complexity (in both hardware and software)

Multiple issue – Starting multiple instructions each cycle. A 4 GHz, 4-way multiple issue processor can theoretically achieve 16 BIPS

Static vs. Dynamic Multiple Issue

Static Issue: Compiler schedules packets of instructions. Each instruction has multiple "issue slots" that are filled with "regular" instructions

In our example, LEGUS architecture, we'll discuss the following:

Each packet has two issue slots,

1) one for an ALU or branch instruction;

2) the second can take a load or a store.

There may be no dependencies within a packet. If one instruction cannot run which some other available instruction, then the compiler must issue a NOP in the other issue lot.

These are effectively 64-bit instructions; hence the name "VIEW" (very long instruction word)

Dynamic Issue:

HW decides which instructions to issue each cycle.

```
char a[1000];
for (i=0; i<1000; i++){ // CBZ, SUBI, ADDI
    a[i]++ ;            // LDUR, ADDI, STUR
}                       // B
```

Instructions of bookkeeping per iteration

Instructions of actual work per iteration

10.12

```
for (i=0; i<1000; i++){      // CBZ, SUBI, ADDI
    a[i]++;                  // LDUR, ADDI, STUR
    a[i+1]++;                // LDUR, ADDI, STUR
    a[i+2]++;                // LDUR, ADDI, STUR
    a[i+3]++;                // LDUR, ADDI, STUR
}                            // B
```

4 instructions of bookkeeping per iteration

12 instructions of actual work per iteration

```
int a[1000];
for (i=0; i<1000; i++){
    a[i]++;
}
```

x0 -> i

x3 -> index (i * 8)

x4 -> Indexed address (a+i == &a[i])

x1 -> base address (a)

```
for:                          // A/B    L/S
SUBI x2, x0, #1000
CBZ  x2, end
LSL  x3, x0, #3
ADD  x4, x1, x3
LDUR x5, [x4, #0]
ADDI x5, x5, #1
STUR x5, [x4, #0]
ADDI x0, x0, #1
B    for
end:
```

Unrolling one time : 8 ALU/Branch , 4 Load/Store

```
SUBI / NOP
CBZ / NOP
LSL / NOP
ADD / NOP
LDUR1 / NOP
ADDI1 / LDUR2
ADDI2 / STUR1
ADDI3 / STUR2
B / NOP
```

12 instructions in 9 cycles

10.14

Memory Organization – Built around the principle of Locality

principle of Locality – Programs access very a small

Temporal Locality –

Spatial locality –

Static RAM Less than a few

cost a few thousand dollars per GB

Dynamic RAM

10/16

inst: 0110101100 ···· 0110

opcode6 = (inst >> 26) & 32-6 & 0x3f

00000 ··· 011010

& 0 ····· 0001111111

011010

opcode11 = (inst >> 21) & 0x7ff

LSL, LSR

R: opcode Rn shamt Rm Rd

31 21 20 16 15 10 9 5 4 0

shamt = (inst >> 10) & 0x3f

CHAPTER 4 4.2 Logic Design Conventions

The datapath elements in LEGv8 implementation consist 2 different type of logic elements

elements that operate on data values are combinational means: outputs depend only on current inputs

combinational elements example: ALU

elements that contain state

An element contains state if it has some internal storage

State elements. If we saved and restored the state elements, it would be as if the computer had never lost power

A state elements has at least 2 inputs and 1 output

One of logically simplest state elements is D-type flip-flop, which exactly has 2 inputs (value & clock) and 1 output

The clock is used to determine when the state element should be written; a state element can be read any time

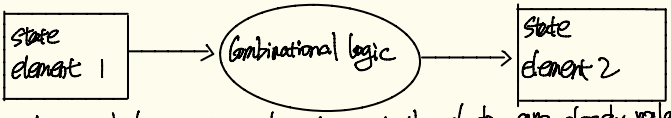
Logic components that contain state are called sequential, because outputs depend on both inputs and the contents of the internal state.

Clocking Methodology

A clocking methodology defines when signals can be read and when they can be written.

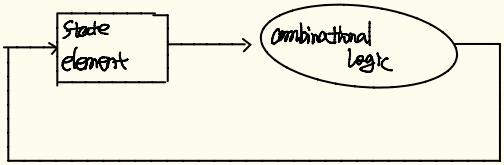
An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa

Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements.



operates in a single clock cycle.

Combinational logic, state elements, and the clock are closely related.



An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values

4.3 Building a Datapath.

examine the major components required to execute each class of LEGv8 instructions

To execute any instruction, we must start fetching the instruction from memory first.

To prepare for executing next instruction, increment the program counter (PC) so that it points at next instruction

PC根据指令地址找到对应指令后，指令就向Register File，R-format needs to 读取2个寄存器，I-format 1个

read two registers.

执行R-format指令时，Read data 1与Read data 2，即两个寄存器的值，作为ALU(Arithmetic and Logic Unit)的inputs

执行I-format指令时，一个input来自经过Sign-extend后的常量

A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. : contains the register state of the computer

We will need an ALU to operate on the values read from the registers.

The CBZ instruction has two operands, a register that is tested for zero, and a 19-bit offset used to compute the Branch Target Address relative to the branch instruction address

CBZ x1, offset.

To implement this instruction, compute the branch target address by adding the sign-extended offset field of the instruction to the PC

4.4 A Simple Implement Scheme

The ALU Control

The LEGv8 ALU in Appendix A defines the six following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

Depending on the instruction class, the ALU will need to perform one of these first five functions.

For load registers & store register

Use ALU to compute the memory address by addition

For the R-type instructions

ALU needs to perform one of four (AND, OR, sub, add) depending on value of 11-bit opcode field in instruction

For compare and branch zero

ALU just pass the register input value

10/19

SRAM....

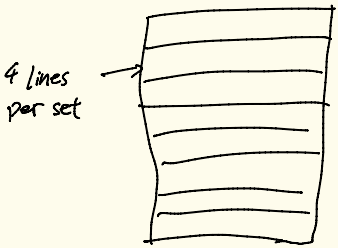
DRAM

Magnetic Disk - A few 10s of dollars per TB

Flash Disk

10.21

associative caches



10/23/2020

cache example 8

Reliability of memory system.

24 kB, 3-way, 32 byte lines, 64 bit each

How many lines are there? How many lines in a set?

Normally take this value and convert to the power of two

divide by 8

3 sets, 8 kB each, 2^5 bytes/line

Total lines: $3 \times (2^{13}/2^5) = 3 * 2^8$

$2^{10}\ 2^3$ 2^{13} bytes/set $2^{13}/2^5 = 2^8$ sets

tag	index	offset
51	8	5

64-13

$(51+1) * 3 * 2^8$ bits $= 52 \times 3 \times 2^5$ bytes $= 3892$ bytes

---

Direct mapped code w/ 6 offset bits, 11 index bits, 64 bit

tag	index	offset
47	11	6

64-11-6

lines are $2^6 = 64$ bits

lines is $2^{11} = 2048$

Code is $2^6 \times 2^{11} = 2^{17} = 131072$

4-way, 5 offset bits, 9 index bits, 64-bit architecture

32 bits lines 2^5

2^9 (512) sets

$2^9 \times 2^2 \times 2^5 = 2^{16}$ bytes whole cache $= 65536 = 64$ kB

4-way

tag	index	offset
50	9	5

Dependability of memory systems

MTTF - Mean time to Failure
MTTR - Mean time to repair

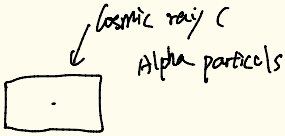
Availability - MTTF / (MTTF + MTTR)

"Nines of availability"

One 9 - 90% → 36.5 day/year downtime
Two 9s - 99% → 3.5 days/year
Three 9s - 99.9% → 9 hours/year
Four 9s - 99.99% → 1 hour/year
Five 9s - 99.999% → 6 mins/year

Non-additive

AFR - Anual Failure Rate



Error Detection and Correction

Parity - Detects 1 bit of error

Does not correct

1 bit per "some amount of memory"

That bit has a value of 0 if the number of 1s in the protected region is even; otherwise its 1

upon detects error, signal is raised to OS

ECC - Error checks codes

Detect x bit errors
Repair y bit errors

- Detect x bits of error, repair y bits of error
  $y < x$
  Typically $x=2, y=1$

Hamming codes: $x=2, y=1$

- A common ECC algorithm is "hamming code"

Reliable of memory

10/26

Hamming codes

0110101 0

value we're storing

	1	2	3	4	5	6	7	8	9	10	11	12	
			0		1	1	0		1	0	1	0	
1	*		*		*		*		*		*		X
2		*	*			*	*			*	*		X
4				*	*	*	*					*	V
8								*	*	*	*	*	V

100011001010

suppose we have cosmic ray

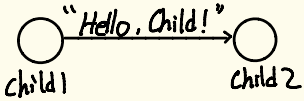
100011001010 => 101011001010 1 bit is wrong

100011001010

101011001010

1110101 1

	0	1	1	1	1	1	0	1	1	0	1	1
	1	2	3	4	5	6	7	8	9	10	11	12
1	*		*		*		*		*		*	
2		*	*			*	*			*	*	
4				*	*	*	*					*
8								*	*	*	*	*


"Hello, Child!"
Child1
Child2

rarchied Page tables

11/04

Inverted Page Tables

Instead of

Page table Entry –

physical #	PID	next index
21	16	21