EECS 370 Final Cheat Sheet

ISA

- `RISC` : Reduced instruction set computing
- `CISC` : Complex instruction set computing

ARM, MIPS & LC2K are both RISC. All RISC instructions have the same length (eg: 32-bits for LC2K). Intel x86 has CISC interface, with RISC core.

MIPS

`add $s1, $s2, $s3`	\$s1 = \$s2 + \$s3
`beq $s1, $s2, 25`	if(\$s1==\$s2) pc=pc+4+100
`lw $s1, 20($s2)`	\$s1 = Memory[\$s2 + 20]
`lui $s1, 20`	\$s1=20*216

Sign extend if signed, and loading half-word or byte. If 0x100 = 00, 0x101 = FF, lbu \$s1, 100(\$s0), then \$s1 = 0x000000FF.

LC2K

`add regA regB destReg`	destReg=regA+regB
`nor regA regB destReg`	destReg=nor(regA, regB)
`lw regA regB field`	regB =*(regA+offset)
`sw regA regB offset`	*(regA+offset)=regB
`beq regA regB offset`	if(regA==regB)pc+=(offset+1)
`jalr regA regB`	regB=pc+1;pc=regA;
`halt`	stop program
`noop`	nothing

Endianess

- Big-endian: most significant bits stored first
- Little-endian: least significant bits stored first

Two's complement

Most significant bit is negative, all else is positive.

$$10000001_{\text{two}} = -1 \cdot 2^7 + 1 = -127$$
$$00000001_{\text{two}} = -1 \cdot 2^0 + 1 = 1$$

To switch signs, invert and add 1. Also assume unlimited sign extension of most significant bit (eg: $1001_{\text{two}} = \cdots 111001_{\text{two}}$)

C to Assembly

Memory layout

Golden rule

Address of variable is aligned based on size of the variable.

- `char` is byte aligned (any addr)
- `short` is half-word aligned (least significant bit == 0)
- `int` is word aligned (two least significant bit == 0)
- `pointer` is size of computer's architecture

Structs

- starting addr % `size(largest basic type)` == 0
- size % `size(largest basic type)` == 0
- allows array of structs

```
struct record {
  float a;        //0-3
  char b;         //4-4, 5-7 padding
  struct {
    char ** c;    //8-11
    short d[3];   //12-17, 18-19 padding
  } s;            //8-19, 12 % 4 == 0
                  //19-23 padding
  double e;       //24-31
  short * f;      //32-35, 37-39 padding
};                //0-39, 40 % 8 == 0
```

Function calls

Pushed & poped on call-frame stack in following order:

1. incoming params
2. \$fp
3. return address
4. callee saved registers (don't save if not needed)
5. local vars
6. spilled registers (when not enough registers)
7. caller saved registers (don't save dead variables)
8. outgoing parameters
9. \$sp

Liveness example

```
int bar(void) {
  int d = 9;     // d: dead,
  printf("1");   //    overwrite on line 4
  d = 5;         // d: alive
  int e = d;     // e: alive
  printf("2");
  return d + e;  // d,e: dead
}
```

Initialization example

```
int g = 0;                     // g: static
void bar(int p, int q) {       // bar: text
  return p + q;                // p, q: stack
}
int foo() {                    // foo: text
  int a = 0;                   // a: stack
  static int b = 1;            // b: static
  void (*func_ptr)(int, int) = &bar;
  // func_ptr: stack
  // *func_ptr: text (points to function)
  int *b_ptr = &b;
  // b_ptr: stack
  // *b_ptr: static (points to global)
  static int *m_ptr = malloc(sizeof(int));
  // m_ptr: stack
  // *m_ptr: heap (dynamic allocated)
  return a;
}
```

Translation

`header`	size of other parts
`text`	machine code
`data`	globals & statics
`symbol table`	symbols & values
`relocation table`	references to variable addresses
`debug info (optional)`	map to source code

`data` does not contain uninitialized data, but keeps track of how much is needed.

`symbol table` contains:

- stuff other files need (globals & funcs defined in this file)
- stuff you need (referenced globals & funcs in this file defined elsewhere)
- static variables

`relocation table` contains:

- address the moved instruction
- type of instruction (lw, sw, jal, etc)
- referenced symbol

```
1  int brown_dog = 656;
2  short black_dog = 343;
3  extern void play_fetch(void);
4  extern int my_dog_age;
5
6  int main() {
7    if(my_dog_age == 0)
8      static int squirrel = 0;
9    my_dog_age = black_dog;
10   int i = 0;
11   while(i < 3) {
12     play_fetch();
13     I++;
14   }
15
16   int dog_treat = 3;
17   int dog_treat_bag = dog_treat * 55;
18   my_dog_age = brown_dog;
19   return 0;
20 }
```

Symbol Table		Relocation Table		
`squirrel`	Data	8	`store`	`squirrel`
`brown_down`	Data	9	`store`	`my_dog_age`
`black_dog`	Data	9	`load`	`black_dog`
`play_fetch`	Undefined	12	`branch`	`play_fetch`
`my_dog_age`	Undefined	18	`store`	`my_dog_age`
`main`	Text	18	`load`	`brown_dog`

Loader

- Creates large enough address space for program to hold text, data, stack
- Copies instructions & data from executable file memory
- Initializes registers (PC and SP most important)

Overview

1. Compiler: *.c →*.s (Assembly)
2. Assembler: *.s →*.o (1st pass: Machine code, 2nd pass: Label resolution)
3. Linker: *.o →*.o, resolves absolute addresses
4. Loader: executes result

Floating math

$$\begin{aligned} 596.75 &= 59675/100 = 2387/2^2 \\ &= 100101010011_{\text{two}} \cdot 2^{-2} \\ &= 1.00101010011_{\text{two}} \cdot 2^9 \\ &= (-1)^0 \cdot 2^{(136-127)} \cdot (1 + 0.00101010011) \\ &= (-1)^{\text{sign bit}} \cdot 2^{\text{exponent}-127} \cdot (1 + \text{mantissa}) \end{aligned}$$

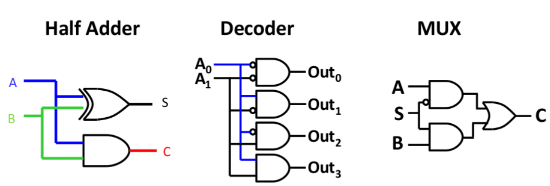
Addition is also the same, raise lower exponent to higher exponent. (Lose least significant bits)

Processor design

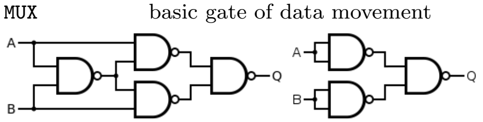
ROM bits $= 2^{\#\text{ of input bits}} \cdot \#$ of output bits

Combinational logic

Combinational circuits

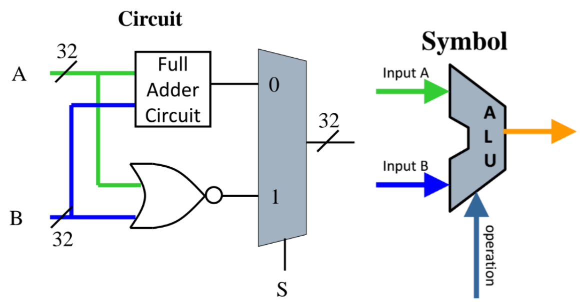


`Half Adder`	basic gate of ALU
`Decoder`	basic gate of indexing
`MUX`	basic gate of data movement



XOR OR

LC2K-ALU



For LC2K, operation may either be add or nand.

Sequential logic

Transparent D-latch

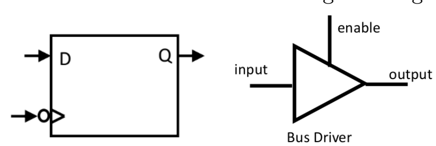
Q is set to D only if gating signal (G) is 1, else the original value is maintained. In finite-state machines, think of D as next state and Q as current state.

Edge triggered D flip-flop

`positive edge`	edge moving from 0 to 1
`negative edge`	edge moving from 1 to 0

Made using two d-latches, with direct/inverted clock signal going to two latches's gating signal.

`inverter on first latch`	positive edge activated
`inverter on second latch`	negative edge activated



Tri-state logic

Three possible states: one, zero, not connected

Finite state machine

$T(s,i)$	current state & input to output state
$P(s)/P(s,i)$	current state to output

Types of FSM:

Moore ($P(s)$)	Output dependent on curr state
Mealy ($P(s,i)$)	Output dependent on curr state & input

Single cycle processor design

1. Fetch instructions
2. Decode instructions (Read data from ROM)
3. ROM output controls data movement (inc. PC, read registers, ALU control)

Each instruction is one cycle, clock drives movement. Clock period is the time it takes to execute the slowest instruction.

Clock speed example

Instr	Instr Mem Access	Read Reg	ALU	Data Mem Access	Write Reg
add	✓	✓	✓		✓
nor	✓	✓	✓		✓
lw	✓	✓	✓	✓	✓
sw	✓	✓	✓	✓	
beq	✓	✓	✓		
jalr	✓	✓			✓
noop	✓				
halt	✓				

Read memory	4 ns
Write memory	3 ns
Read register file	5 ns
Write register file	4.5 ns
ALU	4.5 ns
Others	0 ns

Then slowest instruction would be `lw`, since

Read instruction	4 ns
Read `regA` & `regB`	5 ns
ALU `regA` + `offset`	4.5 ns
Read resulting memory	4 ns
Write `regB`	4.5 ns
Total	22 ns

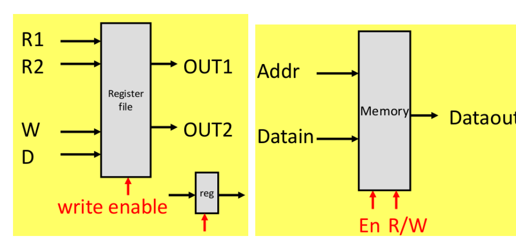
Control building blocks

`MUX`	Select either one of input
`Decoder`	Map instruction to ROM address

Compute building blocks

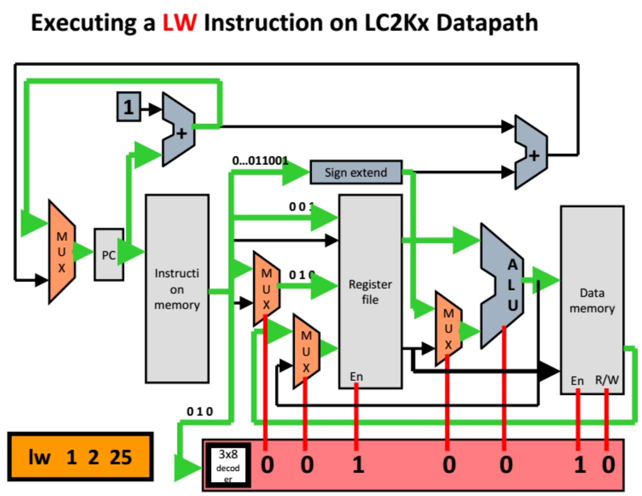
`ALU`	Basic arithmetic operations (ADD or NAND)
`Sign extension`	Repeats most significant bit to output size OUT(31:0) = SE(IN(15:0)) OUT(31:16) = IN(15) OUT(15:0) = IN(15:0)

State building blocks



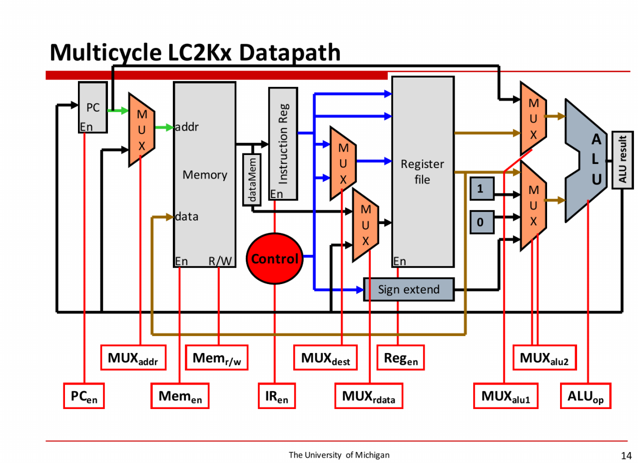
R1, R2, W are 3 bits each, specify OUT1 & OUT2.

LC2K datapath example

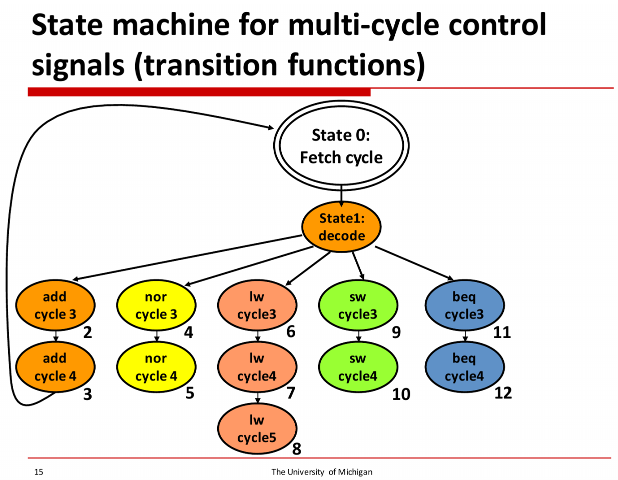


Multi-cycle CPU

Multi-cycle datapath example



Multi-cycle FSM example



FSM example

0. Fetch Cycle
   (a) mem[PC] → instruction register
   (b) Calculate PC + 1
1. Decode
2. ALU
   (a) `regA` + `offset` for lw
   (b) for beq, do equality check also
3. Read memory (Can store in a temporary data mem)
4. Write back (Write actual value to register)

Performance metrics

- Response time (Execution time)
  1. Execution time = total instructions × CPI × clock period
  2. CPI = avg number of clock cycles per instruction for an application, For multicycle we need
     (a) Cycles necessary for each type of instruction
     (b) Mix of instructions executed in the application
     (c) Calculate new CPI from baseline CPI
- Throughput (work/time)

Pipelining... *for dummies!*

Hazards

- Data: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if it is about to be written.
- Control: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- Exception: How do you handle exceptions in a pipelined processor with 5 instructions in flight?

Data hazards

- Avoid: Insert noops
  - not portable
  - large programs
  - CPI is 1, but execution is slower due to noops

```
add   1   2   3    write 3 in cycle 5
noop
noop
nand  3   4   5    read 3 in cycle 5
```

- Detect & Stall: CPI increases
  - pass noop to execute
  - stall at decode, if RaW dependency
- Detect & Forward: May still need stalling.

```
lw  3   6   10    reg 6 at MEM stage
sw  6   2   10    stall 1 cycle
```

Control hazards

- Avoid
  - not portable
  - large programs
  - CPI is 1, but execution is slower due to noops
- Detect & Stall: CPI increases
  - if opcode is branch, pass noop to decode
- Speculate & Squash
  - assume not equal
  - squash at write-back, draw table
  - send noop to memory, execute, decode
  - branch direction prediction using FSM

Cache Magic

- Temporal locality: if you access a memory location (e.g., 1000) you will be more likely to re-access that location than you will be to reference some other random location
- Spacial locality: if we reference a memory location (e.g., 1000), we are more likely to reference a location near it (e.g. 1001) than some random location

Cache avg latency

- Avg access latency = cache latency × hit rate + memory latency × miss rate
- Cache: 1 cycles (90% hit)
- Memory: 36 cycles (99% hit)
- Disk: 400 cycles (100% hit)

Latency: $1 + 0.1 \cdot (36 + 0.01 \cdot 400) = 5$

Stores?

- In cache? Send it to cache.
  - write-through: also send it to memory
  - write-back: mark it dirty, write on evict
- Not in cache? Write to memory
  - allocate-on-write: bring the memory to cache first?
  - no-allocate-on-write: don't do that

Mapping

- Fully-associative: Any memory location can be copied to any cache line.
- Direct-mapped: Memory block can go into specified cache block. Line index bit size is $\log_2(\#\text{ of blocks})$
- Set-associative: Like directed-mapped but with fewer sets. Couple of blocks make a set. Set index bit size is $\log_2(\#\text{-way})$. # sets = #-lines / #-ways

Misses

- Compulsory: First reference to a block
- Capacity: Run out of space
- Conflict: Block is evicted

How? Three step process!

1. Simulate cache with ∞ size → compulsory misses
2. Fully-associative cache with intended size → any new misses capacity misses
3. Actual cache → any new misses conflict misses

Virtual Memory

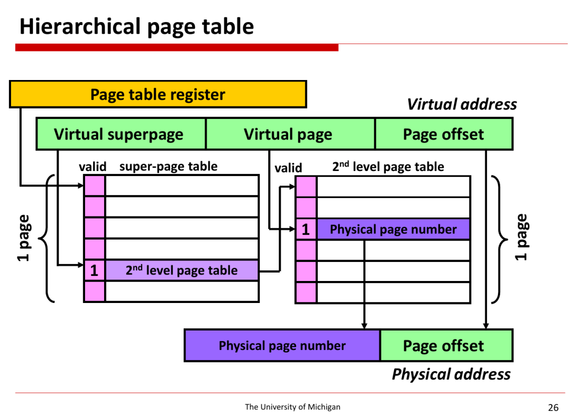
Uses:

- Multiple program can share memory without:
  - transparency: know other programs exist
  - protection: program only access its' memory
- Page fault: V pages misses, handled as exceptions
- Page table: virtual page → physical page, get physical address by physical page with offset

Page table

1GB = 2^{30} bytes of physical memory & 4KB = 2^{12} page size, then the physical page number is 30-12 = 18 bits, plus another valid bit + other useful stuff (read only, dirty, etc.). Approx 3 bytes. How can we organize it?

- Continuous 3MB region of physical memory
- Use a multi-level page table! (Build a hierarchical page table, keep in physical memory only the translations used)
  - Super page table in physical memory
  - Second (and maybe third) level page tables only if needed
  - Size is proportional to the amount of memory used
  - Example: Assuming size is 4KB
    - Min memory used: 4KB (Super page table)
    - Max memory used: $4\text{KB} + 1024 \cdot 4\text{KB}$



How can we replace it?

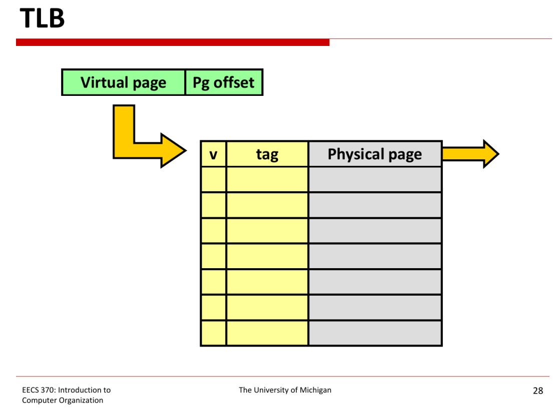
- LRU: Reference bit on page, OS clears occasionally. Evict any unreferenced page.

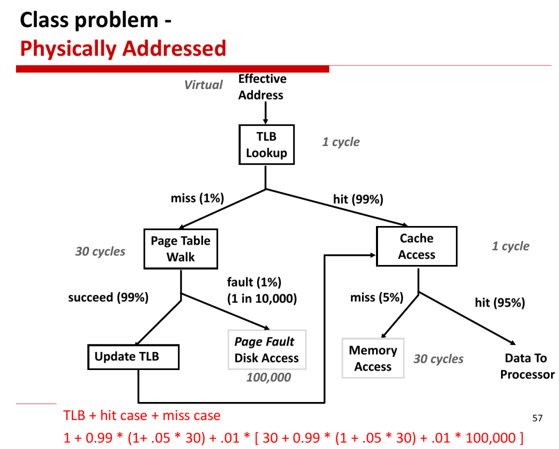
VM cache

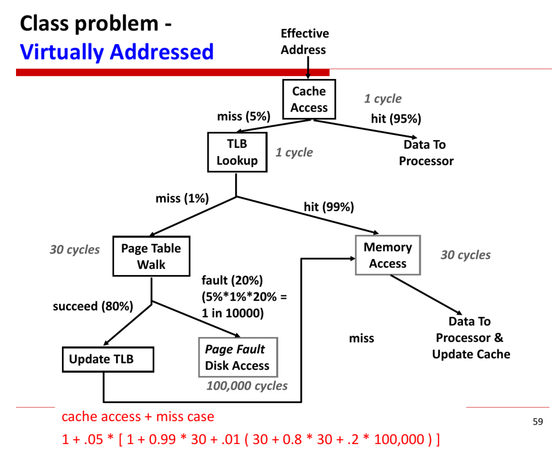
- Virtual addr: Faster access? More complex?
- Physical addr: Delayed access? More complex?

Translation look-aside buffer (TLB)

- Avoid main memory in virtual → physical addr translation (so it's fast)







bang your head against the wall