Class Notes 5/1/2015

Think about recursion

<u>I/O</u>

- eg keyboard input, voltage sensors, monitor output, D/A converter.

General model of device:

- need some way to encode info — today it's unicode & ascii tells you what key is pressed
- need some way to say, "yo, I sent you something"
- sometimes your processor isn't as fast as your devices.

eg keyboard

Explicit IO instructions Need some way to say where do I want data to go, where I want to get it from, etc.

Alternatively,

memory mapping You can just read it from memory

Processor addresses the data register of the keyboard as memory locations

- AKA IO can be read/written to from the address bus as memory

⇒ from Patt & patel for 486
0x00FFFE00 = keyboard status register
AKA ready bit indicates new char recieved

I: KBSR / KBDR } 04 = keyboard data register
AKA last char on keyboard

O: DSR/DDR { 08 = Display status reg.
Device ready for another char
0C = Display Data Reg.
Char. written in low byte of this reg
displayed on te screen

Ideas.
Polling = repeatedly check if it's ready. Asynchronous.
"Are you read? Are you ready?"

Class Notes 5/4/2015
X86; the data path

- "The Pentium Chronicles"
- history: 8086 → i386 → i486 → Pentium → Pentium Pro → Pentium II → 4 → Core
  → core i7
- P.4E added hyperthreading. Threading in software = run 2 progs simultaneously on a single processor.
- Moore's Law: # of transistors doubles every 2 years
  - Became a self-fulfilling prophecy
  - "Throwing a hail mary pass, running downfield, and catching it."
- X86 is a "code museum" b/c preserving backwards compatibility.
- much more built in logic & multiplication
- carry flag added. You can explicitly test & set the flags

How do we build a machine that can execute the instructions?

Stages of processing: (order fake taxonomy)

1. Fetch (instruction)
2. Decode it
3. Execute it
4. Memory (save results)
5. write/back
6. PC update

eg OPl rA, rB

Fetch	icode : ifun	←	$M_1[PC]$
	rA : rB	←	$M_1[PC+1]$
	valP	←	PC + 2

Decode: $\quad$ valA ← R[rA]
valB ← R[rB]

Execute: $\quad$ valE ← valB OP valA
Set CC

Memory
write back: $\quad$ R[rB] ← valE
PC update: $\quad$ PC ← valP
⇨
The DataPath! (OP example)
- eg. Instruction memory "fetches" icode:ifun from PC, "fetches" rA & rB from PC+1 ⇨ we knew here & calculate valP, addr of next instruction
- then Decode, take ~~R~~ rA & rB stuffed into reg. file and get~~s~~ the values
- then pipe values through the ALU
- write back to the registers
- take valP & send it back around to PC

Do the same for other commands...
rrmovl, irmovl, rmmovl, mrmovl, pushl, etc.

Class Notes 5/6/2015

Mrmovl D(rB), rA try

50	rA rB	D

Fetch: icode:ifun $\leftarrow M_1[PC]$
rA:rB $\leftarrow M_1[PC+1]$
valC $\leftarrow M_4[PC+2]$
ValP $\leftarrow PC + 6$

Decode: ~~valA ← R[rA]~~
valB $\leftarrow$ R[rB]

Execute: valE $\leftarrow$ valB + valC

Memory: valM $\leftarrow M_4$[valE]

Write Back: R[rA] $\leftarrow$ valM

PC update: PC $\leftarrow$ valP

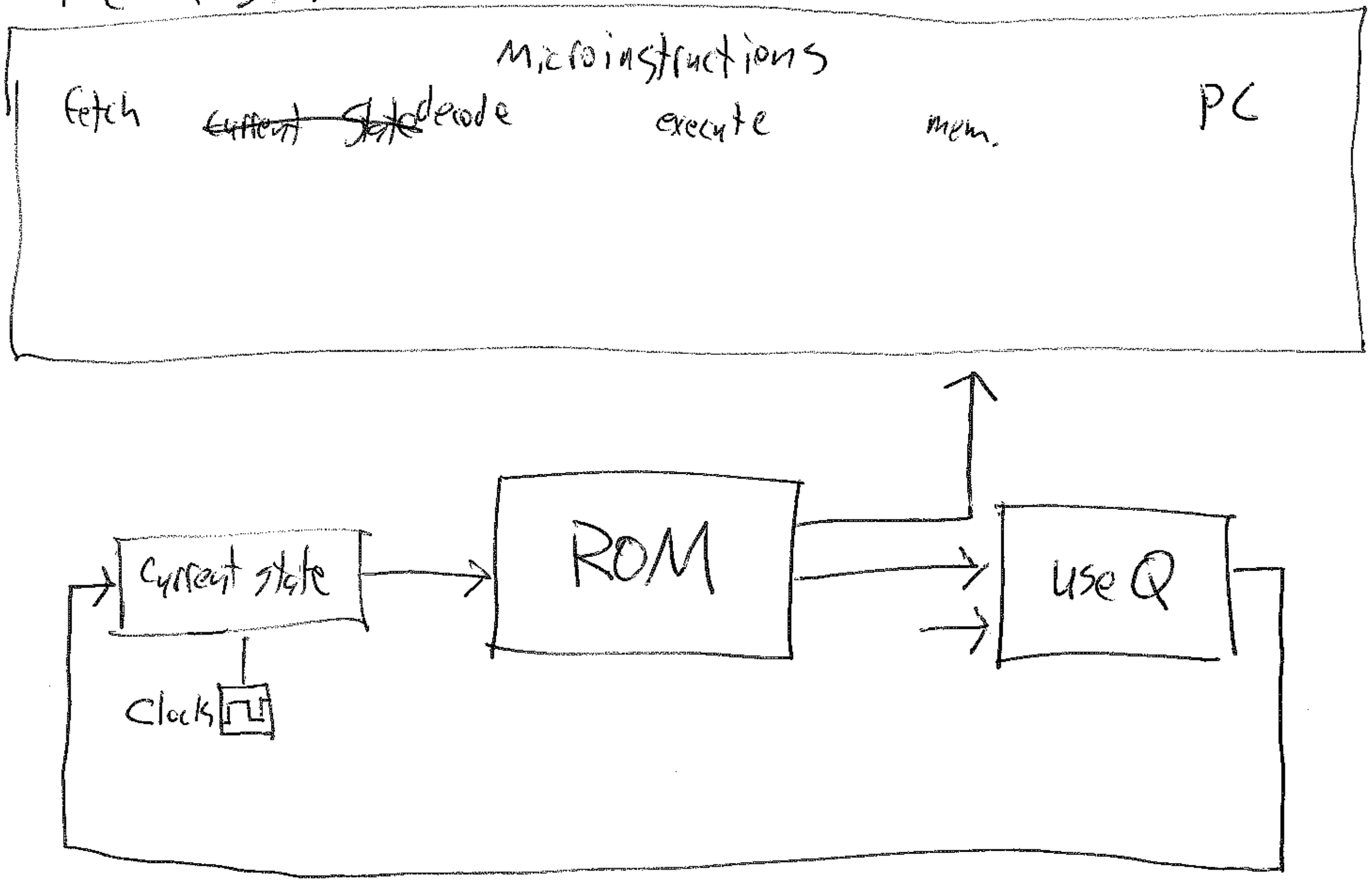
- Note: Pushing the stack pointer pushes the pre-decrementing value of the stack pointer

- No instruction both updates & reads updated value!
  $\Rightarrow$ all w/in one clock!
  Caching & pipelining & hashing kill this.

Class Notes 5/7

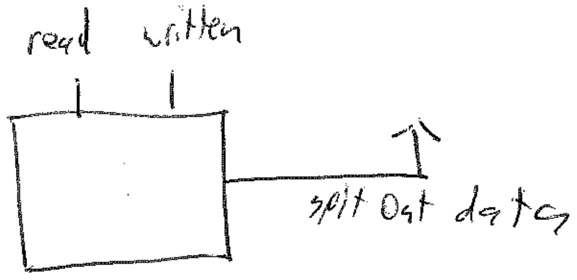
- see the slides to see the microarchitecture of our Y86 processor
- set the values & knobs correctly for each step of the processing process.
- eg rmmovl %eax, 23(%ebx)

The FSM:



- ~~See if to~~ where should the 65K one live?

$\boxed{\text{addr.}}$



For mem.-mapped io,
4-word RAM

- Is the KBSR

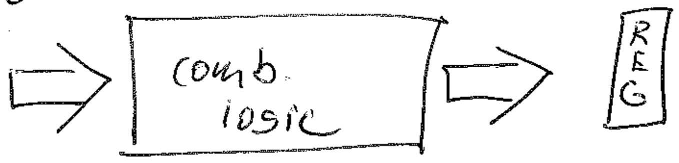
IO

Class Notes 5/8

Pipelining:

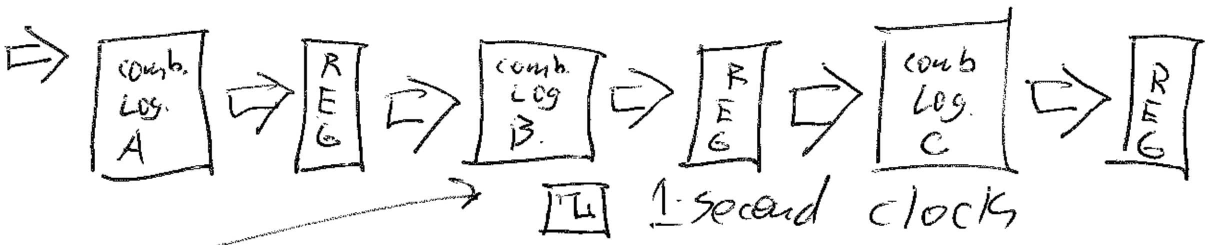
- It takes time to go through a mess of combinatorial logic.
- It would only take about a third as long to get through a 3rd of your logic

w/o



3-second clock

w/



1-second clock

Once you get here, you can set Comb. Log. A to start working on next instruction
eg. split into 5 units on 5 stages of processing

Two ways of measuring time:
- delay (time it takes to do instructions) (AKA latency)
- Throughput (instructions / unit time)

w/ pipelining, delay ↑ (more time for 1 instr.)
But 3X the throughput (3 instructions @ a time)

You can optimize by having more stages for harder processes.
- A super-scalar processor.
- It's a single ~~process~~ stage w/ more processing stages.

Disadvantages:

- non-uniform delays (clock-speed must be as slow as your slowest stage).
  ⇒ waste of time. Life don't chop up easy
- Diminishing Returns
  - Doubling amount of pipelining != doubling throughput
- Data
  - what if next op. depends on the result of your first op.?
    ⇒ fast, but incorrect results
  - Called a hazard. semantics ⇒ changed by pipelining
  - one data soln is "stalling" or we can spy on future result.
  - semantics = what does the execution of this prog. mean?

⇒ Tons of optimization strats using the assembly code in hardware OR compiler.

- pipelining done in runtime & compile time (sometimes)

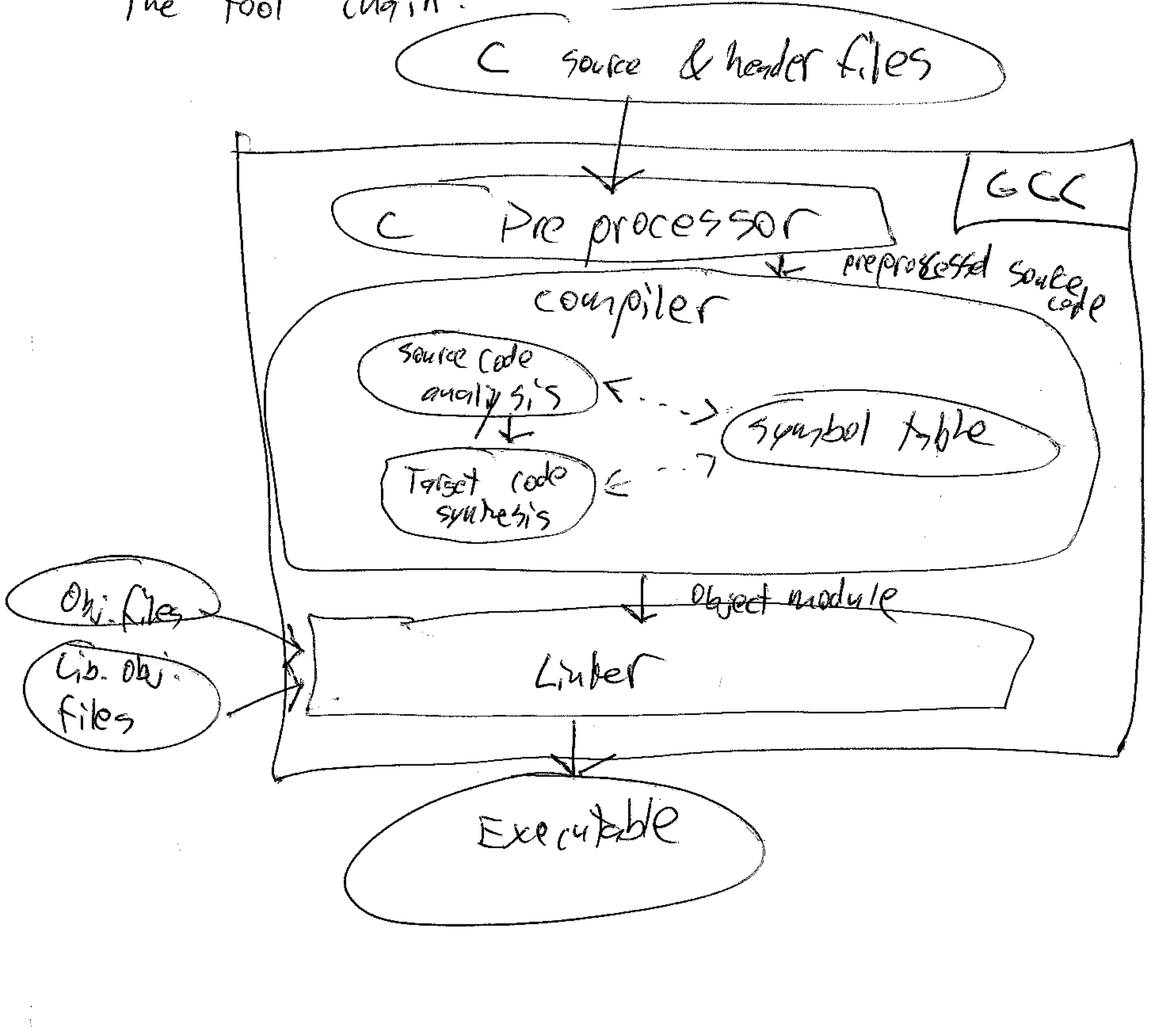
Class Notes 5/8 p.2

- Control Hazard.
  - Value of the zero flag controls ~~pipeline~~ next step
    $\Rightarrow$ you go the wrong way
  - Cause naive pipelining will make it soo you don't know value of ZF.
  - Also, what instruction do we start after a conditional jump??

Class Notes 5/11

- C came after B. lol
  - high-level lang. close to the assembler
- fux w/ yas.c

The tool chain:



Converting C to assembly (at different optimization vals

Push args on to stack before calling...
- maybe return val. on the stack?

Sometimes though, to optimize, store args & return vals in ~~memory~~ registers...

Variables:

- Globals: • visible everywhere • init to 0
  • accessed via some master pointer
- Locals: • visible w/in curly braces
  • Init to ??
  • Accessed via negative offset from %ebp
  • OR kept inside a register.
  • OR optimized away...
- Arguments: • visible w/in function
  • Accessed via positive offset from %ebp

And Scratch
✓

Class Notes 5/13 p.1

- ~~Of~~ "Frames" in de stack!

By convention:

- %eax, %edx, %ecx = caller-save
- %ebx, %esi, %edi = callee-save

IF statements:

1. Get x & y
2. Compare operands
3. Conditional jump to else body
4. ~~Less than~~ If body & non-conditional jump to done
5. else body
6. done

while statements:

1. get conditional var
2. Set initial result
3. Compare c. var to condition
4. Conditional jump to done
5. Body of loop (including changing conditional var & conditional jump to body of loop)
6. Done

For-loop:

```
for (init-expr; test-expr; update-expr)
    body
```

=

```
init - expr;
while (test - expr) {
    body - statement;
    update-expr;
}
```

Logical-and ($x^{\leftarrow \text{in \%ecx}}$ && $y^{\leftarrow \text{in \%edx}}$)

```
1. testl %edx, %edx
2. setne %dl
3. testl %ecx, %ecx
4. setne %cl
5. andb %dl, %cl
6. movzbl %cl, %ecx # ⇒ ecx has the boolean result
```

Conditional assignment: (return x<y ? y-x : x-y;)

1. get x (into %eax ret. val)
2. get y
3. copy y
4. subtract x from y (save else case)
5. subtract y from x into ret val (save if-case as ret val).
6. Conditionally move else-case val into return value

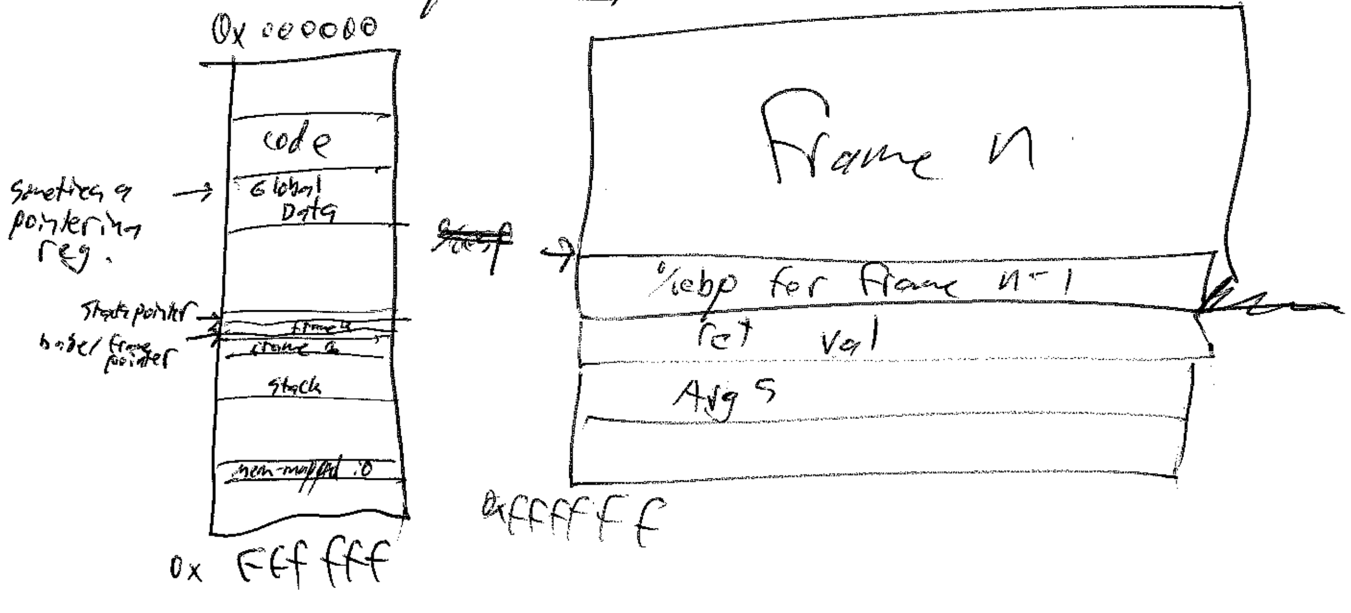
(or just do an if-else structure).

Class Notes 5/13 p. 2

Setting up the frame for functions:

1. push old ebp
2. set current stack pointer to new ebp
3. Subtract space from esp for amount of space you need for local vars ($\Rightarrow$ local vars at positive offset from esp)
4. Do funcion stuff
5. Add space to esp to deallocate the frame
6. pop %ebp (restor old one)
7. return

The address space:



Class Notes 5/18

<u>Caching:</u>

- The data that's in your programmer visible state, takes 0 cycles to get it.
- Data in external RAM... 50-200 cycles...

- Compared to CPU speeds, DRAM / mem speed getting worse and worse and worse
- Caching = have something faster & closer
  ⇒ 1-30 cycles
  - But why not make all of it cached?
    - we can't fit all of mem. here
    - Get static Ram closer

- but how do we figure out what to keep?
  each <u>line</u> consists of
  - <u>Block</u> of B bytes
  - a <u>tag</u>
  - a <u>valid</u> bit
- a <u>set</u> of E lines
- then chache consists of S sets

- Partition address into tag, set index, offset
  use set index to find reset
  use tag to see if any line in that set matches address
  if so, use offset to find the byte w/in the block

6.10

	eg m = address space	C = cache size	B	E (# of lines)	S (# of sets)	t	s	b
1.	32	1024	4	1	256	22	8	2
2.	m	C	B	E	S	t	s	b
	32	1024	8	4	32	24	5	3

$\Rightarrow t + s + b = m$

$C = B \times E \times S$

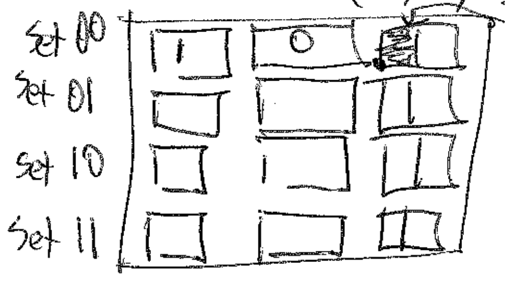
Flavors

1. Direct-Mapped

- $E = 1 \Rightarrow$ one line per set
- multiple sets.

eg

$(S, E, B, m) = (4, 1, 2, 4)$



$m = 4$

can only live here

Addr ~~0000~~	tag bits $(t=1)$	index bits $(s=2)$	offset bits
0000	0	00	0
1000	1	00	0

Same w/ 2 though..
but you need the tag 'cause if tag is 1, that place has mem 1000, but if have tag 0, that place has mem 0000.

Class Notes 5/18 p.2

If valid is zero when you check it, get from mem everything that'd go on that line, set the line's tag.

But that's shitty (cause you get conflict misses)

2.] ~~PD~~ Fully Associative
S=1 $\Rightarrow$ Only one set
- No conflict misses, but expensive AF

3.] Set Associative
multiple sets, multiple lines
• you can still get conflict misses

You get trade-offs, though
- you do start seeing diminishing returns...

Class Notes 5/27

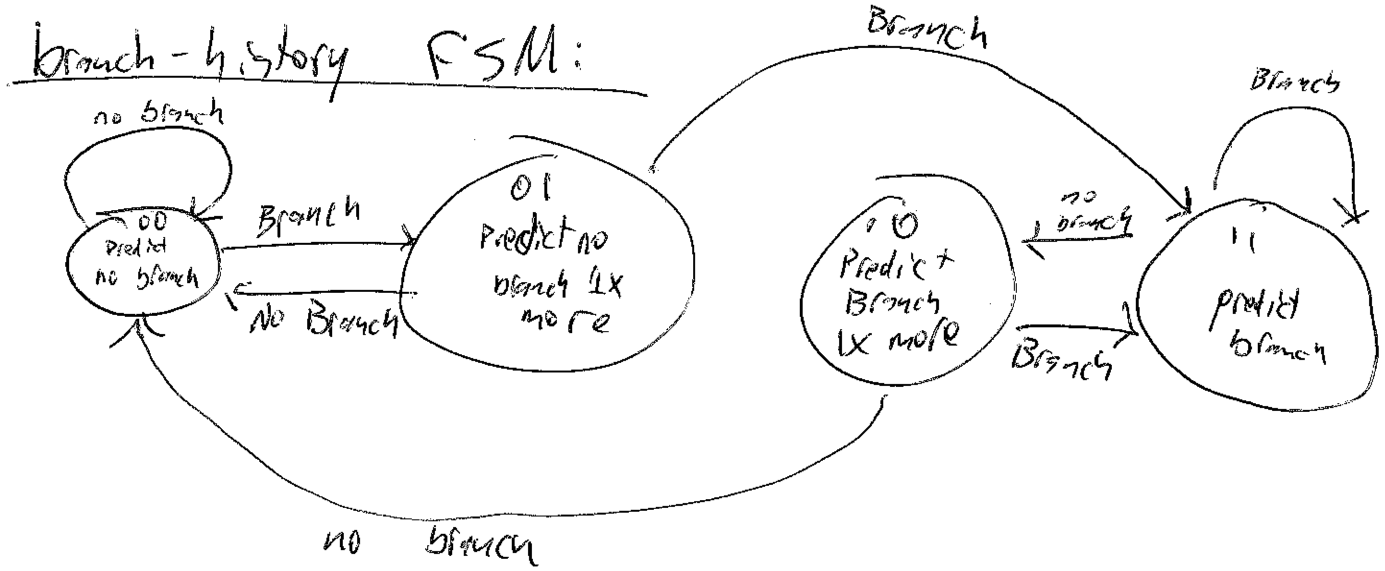
- eg bad.yo

```
xorl                              F D E M W
jne target                          F D E ~~M~~M W
BAD $2, %edx #itarget                 F D ~~E~~ M W
targ +1                                 F D E M W
```

These get turned to nops

- optimizing compilers gets to weird results!
- Branch prediction can be non-trivial

branch-history FSM:



- Processor may juggle instr.s
- but raw / & war dependencies
- See table from tannenbaum
  - Assuming in order → 2 arithmetic op at a time
  - Skipping instr.s → 2 arithmetic ops at a time

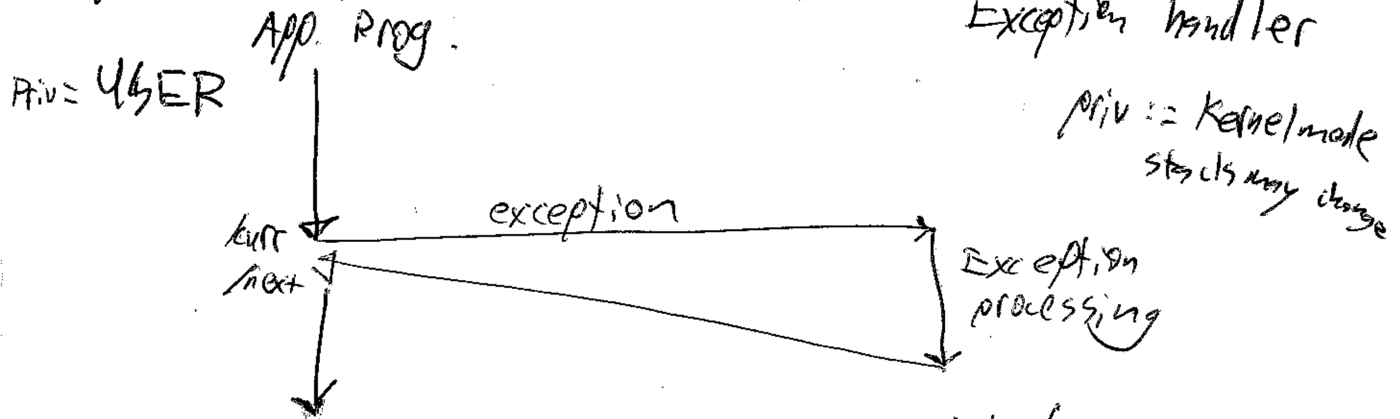
- If you skip instr.s, keep track of what r.s are written to by skipped instrs
- you can write to "secret reg.s" (reg. renaming) when they're being written to

Class Notes 5/29

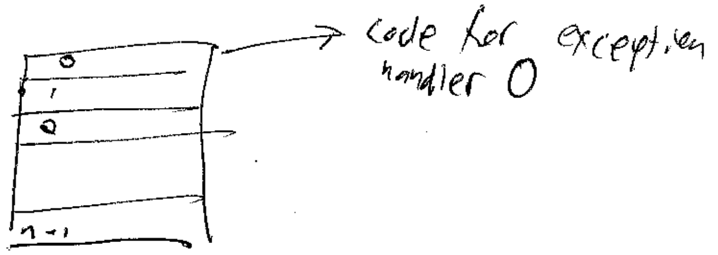
Privilege

- simplest version: storing 1 bit in processor (flip-flop)
  0 == "privileged" (AKA kernel) ⇒ CPU has privilege
  1 == "unprivileged" (AKA user)
- require privilege to do certain, special important ops.
- establish controlled ways in CPU from changing from user → kernel & back
- ring 0 = priv. ring 3 = user

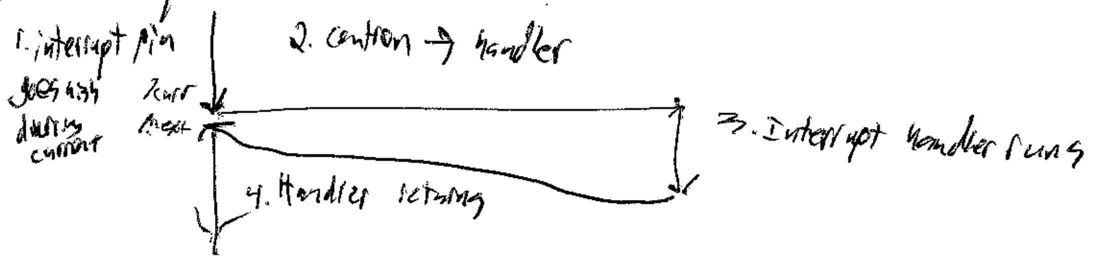
Traps / Interrupts / Exceptions:



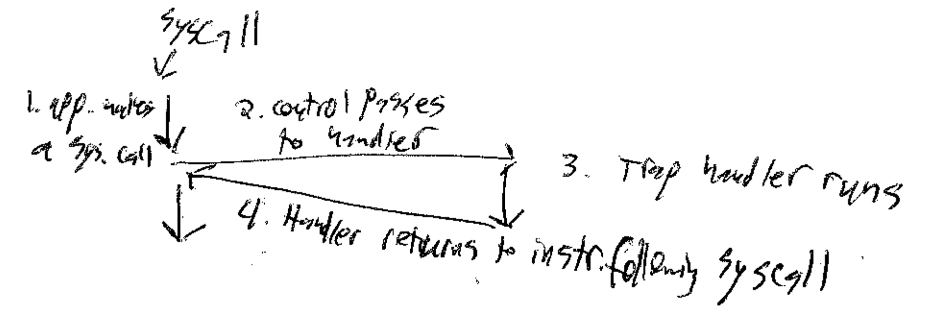
Exception table
= jmp table w/ entry k containing addr. of handler code for exception k



1. Interrupts

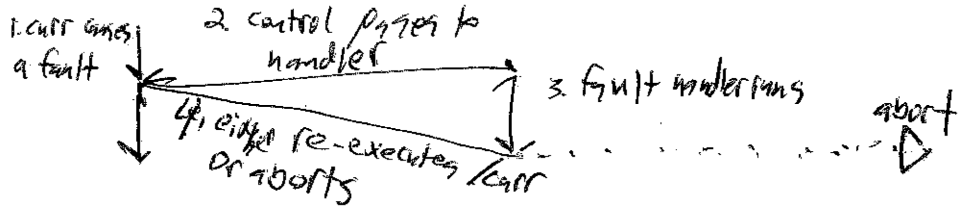

1. interrupt pin goes high during current
Icurr
Inext
2. control → handler
3. Interrupt handler runs
4. Handler returns

2. Trap



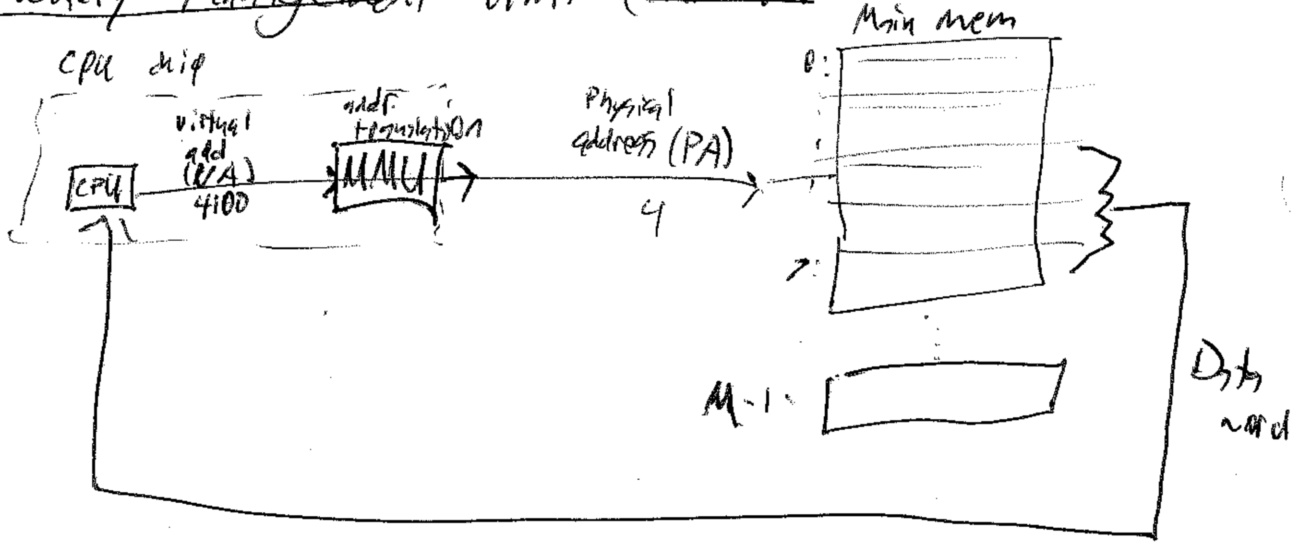
syscall
1. app. makes a sys. call
2. control passes to handler
3. Trap handler runs
4. Handler returns to instr. following syscall

3 Exception



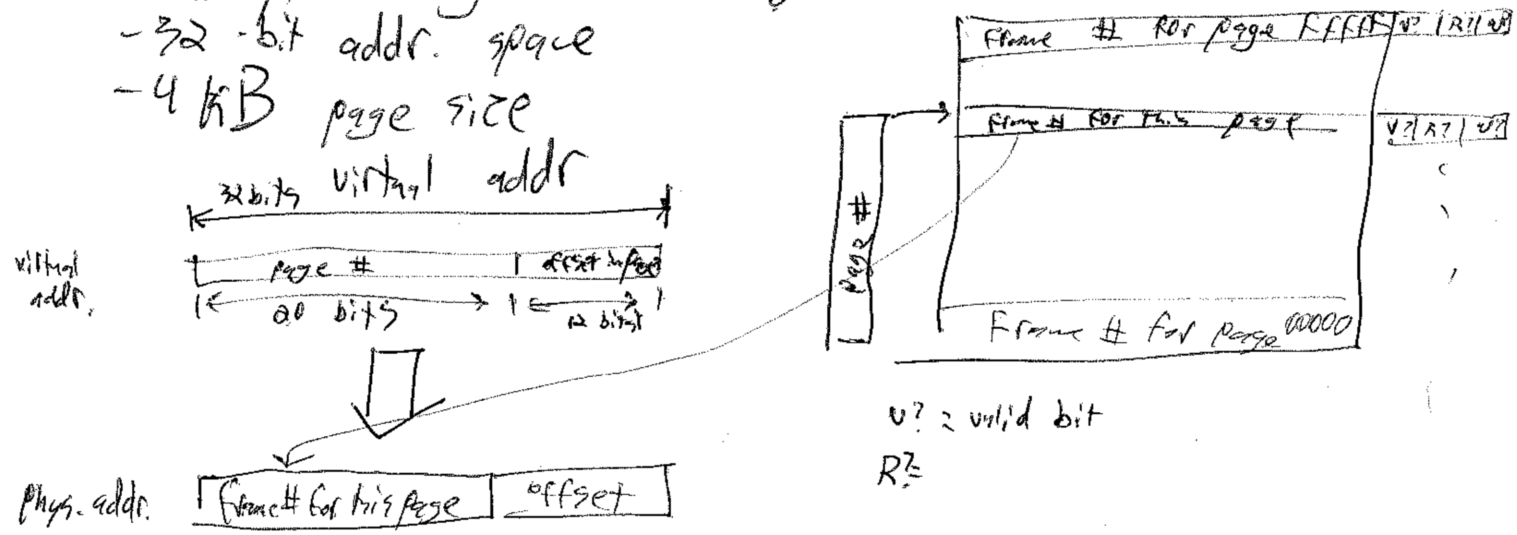
1. curr causes a fault
2. control passes to handler
3. fault handler runs
4. either re-executes curr or aborts
abort

Memory Management Unit (MMU)



Example: Single-Level Page table

- 32-bit addr. space
- 4 kB page size



V? = valid bit

R? =

Class 5/29 P. 2

- ⇨ multiple users sharing memory.
- Each user pointing to at least 1 own frame
- ⇨ can share similar virtual addrs and map to different physical addrs.