

Note part 1: ISA

Lab 1

<<, >> 运算符

>> 表示全体 bit 向右移一位，最高位自动补上 0/1.

关于最高位的处理：如果是 signed (二补码)，那么最高位补上 0 还是 1 取决于最高位是 0 还是 1. 最高位如果是 0 就补上 0 (让正数变小)；最高位如果是 1 就补上 1 (让负数变大) .

note: 往绝对值变小的方向变化！

如果是 unsigned 则最高位自动补 0.

ex: signed 1000 >> 2 = 1110. unsigned 则 1000 >> 2 = 0010

至于 << 左移，不管 signed 还是 unsigned 都是在右边自动补 0. (**note:** 往绝对值变大的方向上变化.)

所以：

```
// 一个 int 中 bit 1 的数量
int numHighBits(int input){
    int count = 0;
    while (input != 0) {
        count += input & 1;
        input >>= 1; // 正数则正常, 负数则死循环
    }
    return count;
}
```

这个程序是错的. 因为当取负数时，这个 input >>= 1 会让 input 永远停在 -1. 它会变成
1,11,111,1111,11111,111111,...

```
// 只能这么写：
int numHighBits(int input){
    int count = 0;
    for (int i = 0; i < sizeof(input)*8; ++i){
        if(input & 1) {
            ++count;
        }
        mask = mask << 1;
    }
    return count;
}
```

sizeof

`sizeof(n)` 表示这个 `n` 的 datatype 占的 bytes 数 (而不是 `n` 这个值本身占用的字节数)

比如 `n` 为 int, 那么 `sizeof(n) = 32` 或 `64`, 取决于语言.

所以拿 `sizeof(n)*8` 就是这个 datatype 占的 bits 数.

```
int numHighBits(int input){  
    int count = 0;  
    for (int i = 0; i < sizeof(input)*8; ++i){  
        if(input & 1) {  
            ++count;  
        }  
        mask = mask << 1;  
    }  
    return count;  
}
```

用 nor 表示 not, and, or

note: nor 只有两个 bit 都是 0 时才是 1, 其他都是 0

所以 $\text{not}(A) = \text{nor}(A, A)$

$\text{and}(A, B) = \text{not}(\text{or}(\text{not}(A), \text{not}(B))) = \text{nor}(\text{not}(A), \text{not}(B)) = \text{nor}((\text{nor}(A, A)), (\text{nor}(B, B)))$

$\text{or}(A, B) = \text{not}(\text{nor}(A, B)) = \text{nor}(\text{nor}(A, B), \text{nor}(A, B))$

Lec 4 - ARM (LEG subset)

ARM arithmetic && logical instructions

(bitwise)

1. ADD, ORR(inclusive), EOR(exclusive) 都是 $x1 = x2 * x3$
2. ADDI, ORRI, EORI 是对应的 I-instructions (第三个 field 是常数): $x1 = x2 + \#immediate(3)$
3. LSL, LSR: logical shift left/right

语法: $x1 = x2 <</>> \#immediate(3)$

这个太重要了 (悲), LSL 还简单就是自己加自己, LSR 就难 implement 了, 有一个现成的 instruction 好多了
逻辑运算都是把 $x2(\text{reg})$ 和第 3 个 field(reg/immediate) 的运算结果存到 $x1(\text{reg})$

ARM memory instructions

我们的便宜 ISA LC2K 的 addressing 方式是 by word (4 bytes in LC2K), 意思是每次 PC+1, 实际上 PC 移动的是向前 4 个 bytes。这意味着我们无法处理 char 等长度小于 4 bytes 的数据类型。 (因而我们的 LC2K 不是功能完备的 ISA)

word 的大小就是一个 instruction 的大小，也就是 ISA 的数据宽度（

64-bit addressing

64-bit ISA 表示寻址范围是第 0 - 2^{64} 个 bytes:

(0x0000 0000 0000 0000 - 0xFFFF FFFF FFFF FFFF)

(Note: 这里这个 8 位 hex 数里面的 1 并不是 bit 而是 byte! !

如果要存储一个 4 bytes 的 int, 那么就是 for example 地址 0x0000 0000 1000 0001 - 0x0000 0000 1000 0004 都是这个 int)

note: 2^{64} 个 bytes 是理论上可达到的 addressing 范围，但是实际上受到内存条等具体硬件的限制。

而 ARM 中一个 instruction 和一个 word 的长度是 4 bytes, 也就是 32-bits, 并不是 64 bits! 64 bits 仅仅代表地址编码的长度，而这个长度和 word 的长度毫无关系。（一个 half word 是 16 bits, 一个 double word 是 64 bits）；

一个 word / instruction / 其他数据类型的长度，代表它们占据几个 bytes, 即一个 object 占据多长的一块地址。

因而如果我们需要往前移动 a int, 我们就要 increment address by 4; 如果要往前移动一个 char, 我们就要 increment address by 1。移动的单位都为 byte.

ARM 中的 regs

ARM 一共有 32 个 regs, 因而在一条 instruction 中要以 5 个 bits 来 encode 一个 reg.

ZXR 表示 R31 寄存器，这是一个零寄存器，存储的值总是 0.

R15 是 PC 寄存器

R30 是 link register

memory instructions (data transfer)

1. LDUR, LDURSW, LDURH, LDURB :

语法: x1, [x2, #simm9]

把 (1) double word; (2) word; (3) half word; (4) 一个 byte 从 [x2 + #simm9] 的 memory 中复制到 x1 reg 上

其中 x2 是一个地址, #simm9 是一个 9-bit signed immediate value (-256~255) 作为 offset, x1 是一个 reg

注意: LDUR 复制 8 bytes, LDURSW 复制 4 bytes, LDURH 复制 2 bytes, LDURB 复制 1 byte.

2. STUR, STURW, STURH, STURB

同理, 只不过是反向复制, 把 x1 reg 上的复制到 [x2 + #simm9] 上

note: 唯一的区别是 LDURSW 的对应是 STURW 没有 S.

3. MOVZ, MOVK

语法: $x1, \#m, LSL \#n$

把某个 4 bytes 的 constant #m 放进寄存器 x1 从第 n 位起左数的 16 位上。

比如 $n = 16$, 那么会把 constant 放到 x1 的 [16,31] 位上.

`MOVZ` 表示把 x1 的其他 48 位全部清零, `MOVK` 表示保持其他 48 位不变.

note: n 只能是 0, 16, 32, 48 中的一个.

处理 load signed/ unsigned word

`LDUR` load 的是完整的一个 8 bytes (64 bit) 和 register 一样长的 word, 所以直接 load 不用管 sign。

`LDURH`, `LDURB` 只处理 Unsigned, 前面全部填上 0.

`LDURSW` 进行了一个 **signed extension**: 对于这个 word 的第 31 位, 如果是 1 那么就前面 32~63 位上全部填上 F, 如果是 0 那么前面 32~63 位上全部填上 0.

ex: 0x7654 3210 被 LDURSW 后是 0x0000 0000 7665 3210

0xF654 3210 被 LDURSW 后则是 0xFFFF FFFF F654 3210

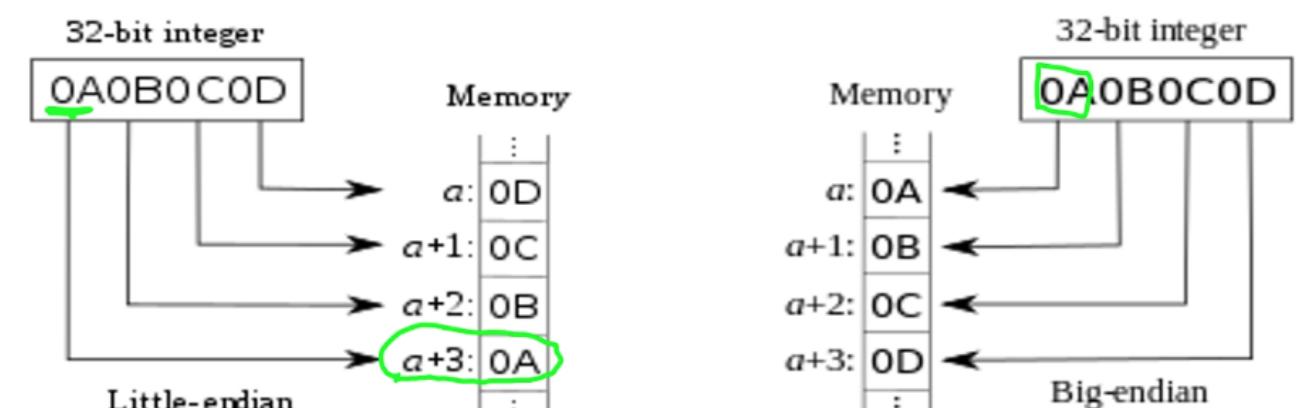
至于 save word: 直接去掉 reg 前面 32~63 位, 自动蕴含了 sign.

所以只有 `LDURSW` 需要考虑 sign.

Big Endian & Small Endian

Endian 表示在一个 half/double/standard word 内, bytes 的 ordering: **significant bits** 的地址在前面还是后面

Little Endian 表示 word 的 4 个 bytes 中从前到后是 insignificant 到 significant; big endian 反过来



ARM 中两种都可以使用, 只是要 consistent, 我们默认使用 little

比如现在我们有两个 word 0xABCD EFGH, 0x1234 5678

那么:

0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	0x1008	
12	EF	CD	AB	78	56	34	12	

Lec 5 - C to Assembly

为了方便（且迅速）read from memory，现代 ISA 要求变量必须是 aligned 的。

Padding for alignment

对于 Primitive object (int, char, etc)

Golden Rule: 对于一个 size 为 N bytes 的 primitive object, 只需要存到下一个 $\text{mod } N = 0$ 的 address 上就可以了。

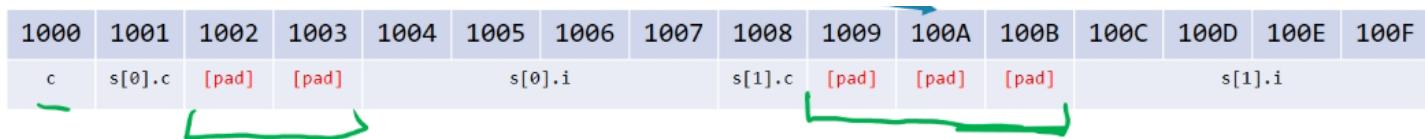
比如现在在 0x1001, 下一个 object 是个 int, 就要跳到 0x1004 上, 中间的 3 格作为 padding 空出来。

对于 sequential object

array: 只需要 treat 每个元素 as independent object 就可以, 一共只需要 padding 一次

对于 non-sequential object

如果只有一个 struct object, 那么看起来只要每个 primitive 成分分开 padding 一下就可以了, 但是我们发现如果我们想要一个 array of struct objects 就会有问题。因为 **beginning address** 的不同会导致这个 **struct array** 中相邻的两个元素之间的 **Padding** 不同, 这样这个 **struct array** 就很难 loop.



解决方法：

除了正常的 Padding 外, 再保证

1. struct 的 starting address 是 struct 中的 largest primitive 的倍数
2. 整个 struct 的 total size 是 struct 中的 largest primitive 的倍数

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B	100C	100D	100E	100F	
c	[pad]	[pad]	[pad]	s[0].c	[pad]	[pad]	[pad]	s[0].i				s[1].c	[pad]	[pad]	[pad]	

note: data padding 是 C compiler 把 C 翻译成 assembly 的时候做的事情。在 struct 外面, 有的 compiler 会 reorder variables to avoid padding, 但是在 struct 里面任何 compiler 都不会 (C99 has forbidden it.)

因而 object 在 struct 内的排布顺序是根据 declare 顺序排序的。所以我们为了省 padding 的空间需要在写 C code 的时候留意一下变量排布。

Control flow (branching)

Sequencing instructions change the flow of instructions being executed.

这是通过用 branching 调整 PC reg 实现的。

ARM branching instructions

Note:

1. branching instructions 比较特殊，并不是 branch by bytes 而是 instructions 的数量，即 **offsetfield** 上是往前多少个 **instructions** 而不是多少个 **bytes**.
2. 不像 LC2K 需要 branch to $PC + offsetfield + 1$ ，ARM 就是直接 **branch** 到 **offset field** 数量的 **instructions**，

Unconditional:

1. B #simm26

PC = PC + #simm26 * 4 位置的 instructions

2. BR Xt

Xt 必须包含了一个 instruction 的地址

PC branch 到去 Xt 这个 reg 包含的地址上面的 instruction.

3. BL #simm26

这是一个带 link 的 branching，通常用于函数调用：会先把 **PC + 4**（本来的下一条 instruction 的地址）储存到 **reg X30 (link register)**，然后跳转到 $PC + \#simm26 * 4$ 位置的 **instructions**.

conditional:

1. CBZ Xt, #simm19

如果 Xt 上的值为 0 则跳转

2. CBNZ Xt, #simm19

如果 Xt 上的值不等于 0 则跳转

3. B.cond #simm19

如果 cond 为 true 则跳转

这一条 B.cond 是比较 high-level 的 implementation，十分智能。我们下面详细讲述

Note: ARM 支持 label 跳转!

ex

```
Again: ADDI X3, X3, #1  
       CBNZ X3, Again
```

B.cond 的用法

我们需要一个 extra status register 存储 condition 的条件

但是这个寄存器并不是我们可使用的 R0 ~ R31 中的一个而是 ARM 的一个特殊的状态寄存器 CPSR

CPSR 中的 NZCV 四个 flags 分别在 31, 30, 29, 28 位

N (Negative flag): 第31位

Z (Zero flag): 第30位

C (Carry flag): 第29位

V (Overflow flag): 第28位

我们可以通过 `CMP`, `CMPI` 来比较两个 register / 一个 reg 和一个 immediate。

```
CMP X0, x1          ; 比较 x0 和 x1  
B.EQ equal_label    ; 如果 x0 == x1, 则跳转到 equal_label
```

在一次 cmp 后比较的结果会被自动存到 CPSR 中

然后我们可以使用 B.cond #simm19 / label 来 branch.

cond 有以下几种:

1. For signed number: B.EQ 表示结果相同; B.NE 表示结果不同; B.LT 表示左边小于右边; B.LE 表示昨天小于等于右边; B.GT,GE 是大于/大于等于;
2. For unsigned numer: B.EQ; B.NE; B.LO; B.LS; B.HI; B.HS

还有特殊的 cond:

我们可以在 **ADD, SUB, ADDI, SUBI** 后面加上 S (ex: ADDS) 来表示 set flag, 把这个运算的结果是否 Negative / zero / overflow / generate a carry

B.MI branch if CPSR 的上个 set flag 的运算结果是负的

B.PL branch if CPSR 的上个 set flag 的运算结果是正/0 的

B.VS / B.VC branch on an overflow set/clear

B.AL always 执行, 等价于 B

Lec 6 - Function call

我们 call function 的时候通常使用 BL, 把 PC + 4 存进 R30 link reg 中, 但是只有一个 Link reg, 而我们如果有多个嵌套函数, 就会损失 return address 的信息

当我们 call function 的时候我们要做这四件事:

1. pass parameters
2. save return address
3. save reg values
4. jump to called function

execute function 后我们要:

5. get return value
6. restore reg values

很显然, 我们的 regs 是 finite 的, 没法把所有 return address, reg values 都放进 regs 里 (并且 data 的大小不一定适合)

所以我们会把这些信息放进 memory 里 (call stack)

和 memory 交换信息处理 data 肯定不如直接在 reg 上处理快, 所以 ARMv8 的 solution 是: 把 first few parameters 放进 regs (X0-X7), 把剩下的放进 memory 的 call stack 上。

Call Stack

ARM 在程序运行中会 allocate a region of memory, 称为 call stack.

Call stack 用以 manage 所有的 storage requirements to simulate function call semantics.

1. parameters (that were not passed through regs)
2. local vars
3. temporary storage (run out of regs 时)
4. return address
5. ...

每次做一个 function call 就会有一个 stack frame 被放上 call stack, 并且这个 stack frame 在 return from function 的时候被 deallocate.

类似于 PC, 我们有一个 **stack pointer(SP, X28)**, **keep track of current top of stack.**

内存布局结构

stack 在最上方，最上部是封死的，新 frame 加入栈顶时，向下增长（栈顶在下面）。

heap 在 stack 的下方，动态内存被分配时，向上增长。

(stack 和 heap 分别朝相反的方向扩展，因此在内存不足或者栈和堆碰撞时，可能会引发 stack overflow)



Static 段存放 global & static variables，在程序 loaded 时就被确定，在整个程序运行期间不变。

Text 段在最底端，read only.

一个程序中，dynamic memory goes to heap，static & global 的变量 goes to static；parameters & local variables go to stack

ex:

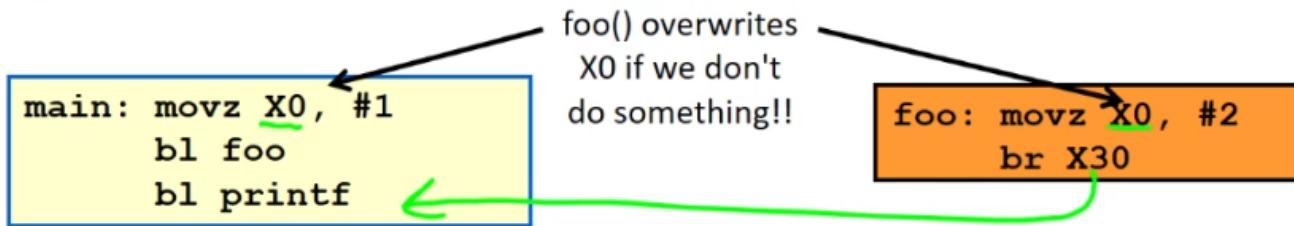
```
int w; // w on static (global)
void foo(int x) { // x on stack
    static int y[4]; // y on static
    char* p;
    p = malloc(10); // p on stack, the 10 btyes on heap
    ...
    printf("%s\n", p); // "%s\n" on static
}
```

note: "%s\n" on static 的原因是这是一个 string literal，不可变，在程序编译时就固定，不论 foo 被调用多少次，它的储存位置和内容都不变。

Saving regs

Assembly 中，所有 functions 都只共享 32 个 (ARM) regs

call function 后我们会 Overwrite regs.



所以我们需要 store reg values.

关于 save reg values 我们有两个办法：

1. **callee saved**: 被 called 的 function 在 **overwrite reg values** 前把它们 **save** 到 **stack** 上，并且在 **return value** 之前 **restore** 它们。

ex:

```
main: movz X0, #1
      bl foo
      bl printf

foo: stur X0, [stack] //save X0 on stack
     movz X0, #2
     ldur X0, [stack] //restore X0
     br X30
```

2. **caller saved**: calling function 在 **function call** 前 **save reg values on** 自己的 **stack**，并且在 **function call** 结束后 **restore** 这些 **regs**.

```
main: movz X0, #1
      stur X0, [stack] //save X0 on stack
      bl foo
      ldur X0, [stack] //restore X0
      bl printf

foo: movz X0, #2
      br X30
```

caller-save 和 callee-save 的优劣势

caller-save 的 must save. 在 call 和 callee() 之间，如果 callee() 里面还声明了 caller() 的变量，那么 call callee() 前必须 store 这些变量用到的 reg.

最小的函数不用 save，因为它不是 caller.

callee-save 的 must save: callee() 的 stack frame 里所有用到的 reg 都必须在它 overwrite 一个 reg 时 save.

最大的 main() 函数不用 save，因为它不是 callee.

所以显然优劣势：

caller-save 适合在 call 发生时它下面的 live variables 不多的情况

callee-save 适合 callee 的 local variable 不多的情况。

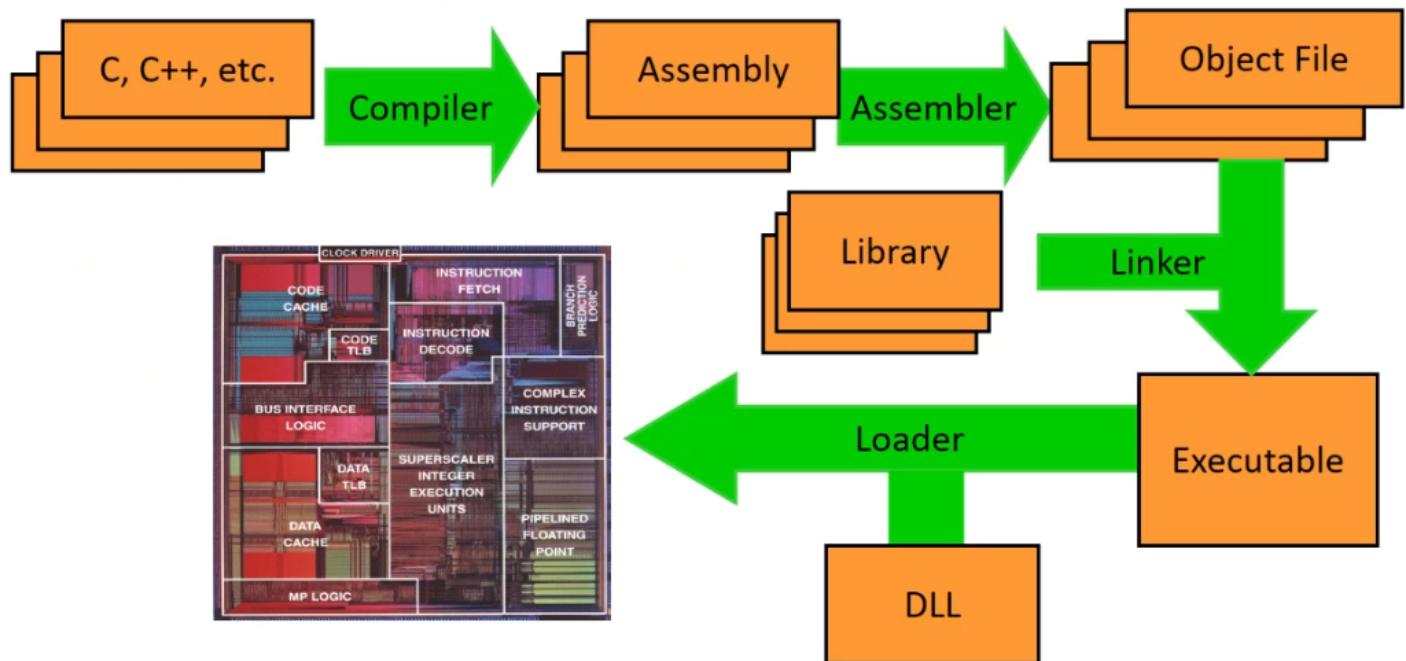
混用可以达到比较好的效果。

convention 上，我们习惯分出 caller-saved regs (通常为0-15) 和 callee-saved regs(通常为19-27).

我们希望在 main 中的变量尽量使用 callee-saved regs，在没有嵌套函数的函数中的变量尽量使用 caller-saved regs.

Lec 7 - Linker

Source Code to Execution



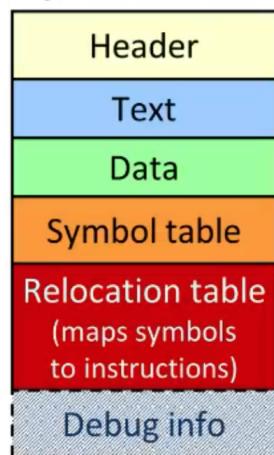
higher-level languages 会先通过 compiler 编译成 assembly,

assembly 再由 assembler 转为 object files

object files 加上 Libraries 再通过 **linker** 转为 exe

Object file format

Object code format



自上到下: Header, Text, Data, Symbol table, Relocation table 以及 Debug Info. Debug Info 只有在 compile with "-g" flag 的时候才被 included.

Note: object file 里面都是 machine code. 这里为了清晰使用 assembly 来直观代替.

Header	Name	foo
	Text size	0x0C //probably bigger
	Data size	0x04 //probably bigger
Text	Address	Instruction
0		LDUR X1, [XZR, G]
4		ADDI X9, X1, #1
8		BL B
Data	0	X 3
Symbol table	Label	Address
	X	0
	B	-
	main	0
	G	-
Reloc table	Addr	Instruction type Dependency
	0	LDUR G
	8	BL B

1. Header: 用来 keep track of 每个其他 section 的 size

2. Text: 就是这个 as 文件的 machine code

3. Data: 相当于 LC2K 里面的所有 .fill

包含所有 initialized globals 和 static locals

Symbol Table

4. Symbol table: 列举了所有能够在这个文件外被看到的 labels. 比如 **function names, global variables** 等.
对于每个 symbol 我们都表明它的 section, 比如 int a 在 data section, 函数 foo 在 text section; 至于 **extern** 的变量和函数的 **section** 我们则留 **blank**, 在它们自己的文件里留 section

ex:

```
extern void bar(int); // bar: extern
extern char c[];
int a; // a: global var
int foo(int x) { // local var x, 并不 visible to 其他文件
    int b; // local var b, 并不 visible to 其他文件
    a = c[3] + 1;
    bar(x);
    b = 27;
}
```

symbol table (assembly 简记):

```
a data
foo text
c -
bar -
```

Relocation Table

5. Relocation Table: 列举所有当 things are moved in memory 时应该被 updated 的 instructions 和 data.

主要负责对全局变量和跨模块引用的函数进行重定位, 而不处理局部变量 (local variables) 相关的指令, 这是由 **局部变量的生命周期、作用域以及存储方式决定的**

因为函数调用期间 local vars 被分配到 stack frame 上, 在函数返回时被释放; 所以 local vars 的地址是相对于 stack point 的偏移量, 在 compilation 时可以确定, 不需要 relocation

而每个函数都可以单独 assemble, 对 **global variables (data)** 以及对其他函数 (**code**) 的引用是跨文件的, 因而 **relative placement of code/data** 在 **compilation** 阶段是未知的 (在 **linking** 阶段才知道), 编译器不知道它们的 **absolute address**, 所以需要 **relocation table** 记录这些符号引用, 在 **linking** 阶段再换为具体地址.

Relocation 需要做的: relocate absolute reference to reflect placement by the linker

(1) PC-relative 的 addressing (beq 等): **never relocate**.

(2) Absolute Address (mov 等): always relocate

(3) External Reference (bl 等): always relocate

(4) Data Reference (movz/movk 等): always relocate

ex:

(Note: c 中任何的 declaration 都不是一个 instruction. 只有给具体的 value, 才会变成 instructions compile 进入 assembly 文件里. 所以)

```
extern int c[];
extern void B();
int x = 3;
int foo() {
    int b;
    x = c[3] + 1;
    B();
    b = 27;
}
```

变成 assembly 的部分是：

file1.c

```
1 extern void bar(int);
2 extern char c[];
3 int a;
4 int foo (int x) {
5     int b;
6     a = c[3] + 1;
7     bar(x);
8     b = 27;
9 }
```

其中我们看到：

1. 第 6 行是一个 load global var c, addition 和一个 store gloval var a 的操作
addition 不需要 relocation. load 和 store 的由于都是 global var 的, 需要 relocation.
2. 第 7 行是一个 function call, 需要 relocation.
3. 第 8 行是 load 和 store 一个 local var, 在 stack 上, stackpointer 在 compilation 时确定, 不需要 relocation.

所以 relocation table:

```
line  type  dep
6    LDUR   c
6    SDUR   a
7    BL     bar
```

Relocation table 仅仅只是 Used by assembler 和 linker 的, .exe 文件不包含任何 relocation info!

Linker: Stitch objs into a single .exe

在 assembly 把 .c 等等文件都做成 .o 之后, linker 把这些 .o 链接成一个 .exe 文件.

Note: Libraries 只是特殊的 object files.

Step:

1. 把所有 .o files 的 text segment 放在一起
2. 把所有 .o files 的 data segment 放在一起, 并 concatenate 它到整合后的大 text segment 的后面
3. Resolve cross-file references to labels. 确保没有 undefined labels.

Loader

.exe 文件被制成品后被放在 disk 上

loader 负责把 .exe 文件的 code image 放进 memory, ask the OS to schedule it as a new process.

具体:

1. create 一个新的 address space to hold text/data segment 以及 along with a stack segment
2. 把 instructions 和 data 从 .exe 文件复制进新留出的 address space
3. initialize regs (PC, SP等)

Note Part2: Digital Logic

Note part 1: ISA

Lab 1

<<, >> 运算符

sizeof

用 nor 表示 not, and, or

Lec 4 - ARM (LEG subset)

ARM arithmetic && logical instructions

ARM memory instructions

- 64-bit addressing
- ARM 中的 regs
- memory instructions (data transfer)
- 处理 load signed/ unsigned word
- Big Endian && Small Endian

Lec 5 - C to Assembly

- Padding for alignment
 - 对于 Primitive object (int, char, etc)
 - 对于 sequential object
 - 对于 non-sequential object
- Control flow (branching)
 - ARM branching instructions
 - B.cond 的用法

Lec 6 - Function call

- Call Stack
- 内存布局结构
- Saving regs
 - caller-save 和 callee-save 的优劣势

Lec 7 - Linker

- Object file format
- Symbol Table
- Relocation Table
- Linker: Stitch objs into a single .exe
- Loader

Note Part2: Digital Logic

Lec 8 (1): 表示 float nums

Lec 8 (2): Combinatorial Logic

- Mux(multiplexor)
- Decoder
- Adder
 - half-adder
 - full-adder
- 从 n-adder 得到 n-substracter

ALU design

Propagation delay

Lec 9: Sequential Logic

- SR latch
- D latch (improved SR latch)
 - 画 D-latch Timing Diagram
- D flip flop (improved D latch)
 - 为什么 Flip Flop 可以避免PC被 increase 多次的问题

Lec 10: Finite State Machine

- ROM(read only memory)
- 计算 size of ROM

Lec 11: Single-Cycle Datapath

- State Building Blocks: reg files, memory
- Overall View
- ADD/NOR
- LW/SW

耗时

BEQ

耗时

JALR

Lec 12 Multi-Cycle Datapath & Pipelining

Multi-Cycle Execution Overview

State 0 (Universal): fetch

State 1: Decode

如何确定下一个 state 是五个里面哪一个?

State 2-3: Add

State 6-8: lw

State 11-12: beq

Multi-cycle behavior

Lec 13 Pipelining

Implementation Idea

Stage 1: Fetch, IF/ID reg

Stage 2: Decode, ID/EX reg

Stage 3: Execute, Ex/Mem reg

Stage 4: Memory Op, Mem/WB reg

Stage 5: Write Back

Overview

Lec 14 Data Hazard

Data Hazard 和 Data Dependency

Method 1: 加入 noop 来避免 data hazard

Method 2: Detect and Stall

compare 具体实现

Method 3: Detect and forward

Data Hazard 的所有类型

处理前四种 Data hazard

处理 sw/lw 的 data hazard

Lec 15 Control Hazard

beq 的流程

Method 1: 在 assembly 层面避免 branch

Method 2: Detect and stall

CPI 分析

Method 3: Speculate and squash-if-wrong

Squash 具体做法

除了 Data Hazard 和 Control Hazard 外的 Exceptions

CPI Calculation Problems

Problem 1: 计算 CPI 和 TPI

Problem 2: Adding a PC Writeback

Problem 3: 10 stage pipeline

解决 hazard 的逻辑总结

Branch Prediction: 更多 Speculation 策略

需要 predict 哪些东西

Predict branch & branch address

predict branch direction

Static branch prediction

Dynamic branch prediction

Note Part 3: Cache and Memory Design

Lec 16 - Cache

Options to represent a bit

Option 1: SRAM(Static Random Access Memory)

Option 2: DARM(Dynamic Random Access Memory)

Option 3: Disks

HDD(Hard Disk Drive)

SSD(Solid State Drive)

Memory Hierarchy

Function of the Cache

temporal locality

example

Cache overhead

如果 Cache filled

Hit/Miss rate

Lec 17 - Improving Caches

Reducing overhead

Spatial Locality

LRU with more than two entries

Iteration is not actually costly

store: write back/through

write-through policy

write-back policy

性能对比

Lec 18 - Direct-mapped cache

map a mem block to a cache line

划分一个 address

Note: direct-mapped cache 不需要 LRU bits

Lec 19 - Set-Associative cache

How set associative cache divide address space

example

Practice problem

More Sophiscated: Ln Cache, I/DCache

Lec 20 - Classifying Cache Miss

3 reasons for cache misses

Practice problem

减少 Miss 的 3 个对应方法

Cache size 的影响

Block size 的影响

Associativity 的影响:

Practice problems

Lec 21 - Virtual Memory

Pages

Page Table

Extend VM to disk

valid bit 的使用

Lec 22 - Multi-Level VM

Size of page table

Multi-Level Page Table

Data Structure
Dividing an address
Lec 23 - Speeding up VM
Table Look-aside Buffers (TLB)
Virtual Memory walkthrough
Placing Caches in a VM System
Physically addressed cache
Virtually addressed cache

Lec 8 (1): 表示 float nums

digital logic 前的最后一个主题：如何表示 float numbers

float number 的表示方法是使用 IEEE floating point format, 和 2's complement 完全不同.

single precision 单精度 (float in c++):

32位数字，其中

1. Most significant 是第 31 位表示正负：0 表示正，1 表示负
 2. **exponent**: 第 23-30 这 8 位是一个 exponential 数字，表示这个数字乘以二的多少次方。它以 **-127** 为偏移量，也就是说这八位数字从 0-255 的范围，可以表示从 -127 到 128. 所以它的范围是 2^{-127} 到 2^{128} 次方。所以可以表示很接近 0 到很大的数
比如 `10000101` 就是 133，于是 $\text{exponent} = 133 - 127 = 6$ ，
 3. **Mantissa**: 第 22 到 0 个 Bits 是 23 个有效数字. 它代表 1 后面的小数部分 (隐含了最前面的一个 1，所以其实是 24 位有效数字)
- M 可以表示 2^{-23} 到 $1 - 2^{-24}$ 的 $(0, 1)$ 之间的精度.

所以最后的结果是 $\pm 1.M \times 2^{\text{exp}}$.

并且我们发现其实 **exponential** 就代表了把小数点后移多少位。

起始的地方在 1.0，第零位是 $2^0 = 1$ ，第一位是 2^{-1} ，第二位是 2^{-2} ，。。。如果 $\text{exponent} = 133 - 127 = 6$ ，那么第一位就变成了 $2^{0+6} = 2^6$ ，第一位变成了 2^5

我们不难发现一件事情就是：当 exponent 变大的时候，我们能表示的两个数之间的间隔就变大了，精度就变小了。比如当 $\text{exponent} = 23$ 的时候，我们就无法精确表示小数了；当 $\text{exponent} > 23$ 的时候，我们甚至无法表示一些整数了。

双精度浮点数 (double in C++) 有 64 位，其中有 53 bits of precision。所以它在 $\text{exp} > 53$ 时也会无法表示整数

也就是说如果我们 declare `double x = pow(2, 55) + 0.23332131;`，我们并无法得到我们想要的数而是被迫只留下 2^{53} 即便使用 long double，还是有同样的问题

解决办法是我们自己写一个数据结构，通过算法来实现无限大精度。 (其实不需要，因为现在有很多这样的 library 比

如 GMP 等)

Lec 8 (2): Combinatorial Logic

一些小 trick:

$$\text{XOR} = \text{not}(\text{A nor}(\text{A nor B}) \text{ nor}(\text{B nor}(\text{A nor B})))$$

$$\text{not}(A) = \text{nor}(A, A) = \text{nand}(A, A)$$

Mux(multiplexor)

Mux: $S = 0$ 时选择 A (上) 的值, $S = 1$ 时选择 B (下) 的值

mux 和 nor, nand 一样都是 universal 的! 一个 2^N entry 的全 truth table 可以通过 N

mux: $S ? B : A$

即 **(A and notS) or (B and S)**

我们可以通过嵌套 mux 来达成多个 Input 的 mux.

(programmable hardware)

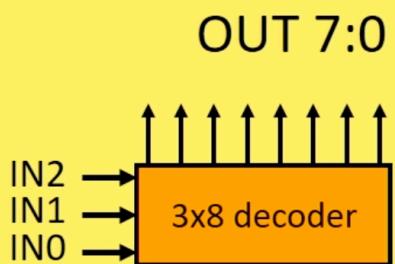
Decoder

一个 decoder 获取一个 n-bit binary 作为 input, 并且 output 2^n 个 Bits. 其中有且仅有一个 bit 是 1, 其他都是 0

比如一个 3x8 的 decoder, 获取一个 3 位的 binary 作为 Input, 这个 binary 的范围是 0-7 (0-indexed), 于是我们的 8 位输出中对应这个数字大小的一位是 1, 其他是 0.

ex: $0b101 = 5$, 所以 decoder 的第六位 (b5) 是 1.

3 to 8 decoder



Read-only Memory



Decoder activates one of the output lines based on the input

IN	OUT
210	76543210
000	00000001
001	00000010
010	00000100
011	00001000
etc.	

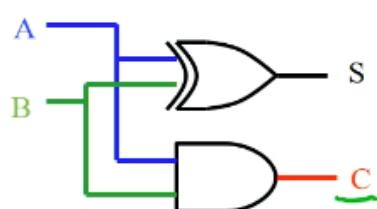
ROM stores preset data in each location

- Give address, get data.

Adder

half-adder

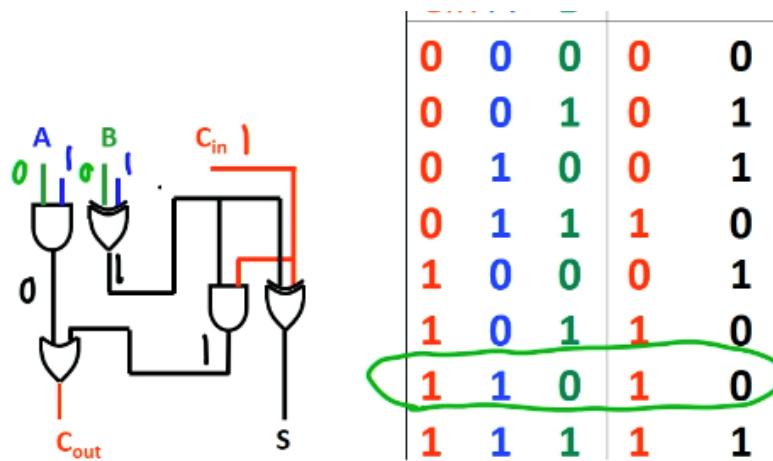
half-adder 不考虑先前的进位，是一个简化的计算一个 bit 和一个 bit 相加的工具。我们获取两个 bits，算它们相加后这一位的结果和是否有进位（一共也就 0, 1, 2）



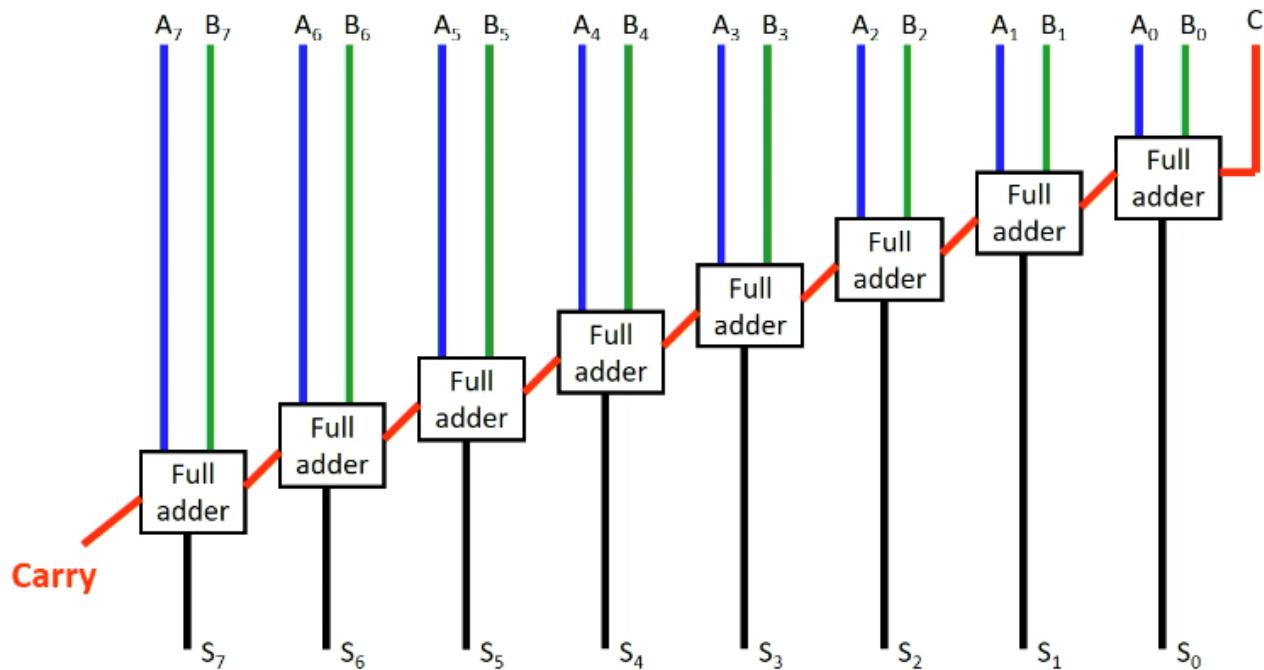
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

full-adder

我们可以增加一个 Input: 上一位的 carry bit, 这样就可以得到一个 full adder.



通过连接 n 个 full adder, 我们可以得到一个 n -bit adder



This will be very slow for 32 or 64 bit adds, but is sufficient for our needs

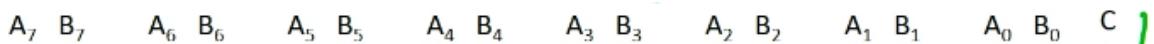
从 n-adder 得到 n-substracter

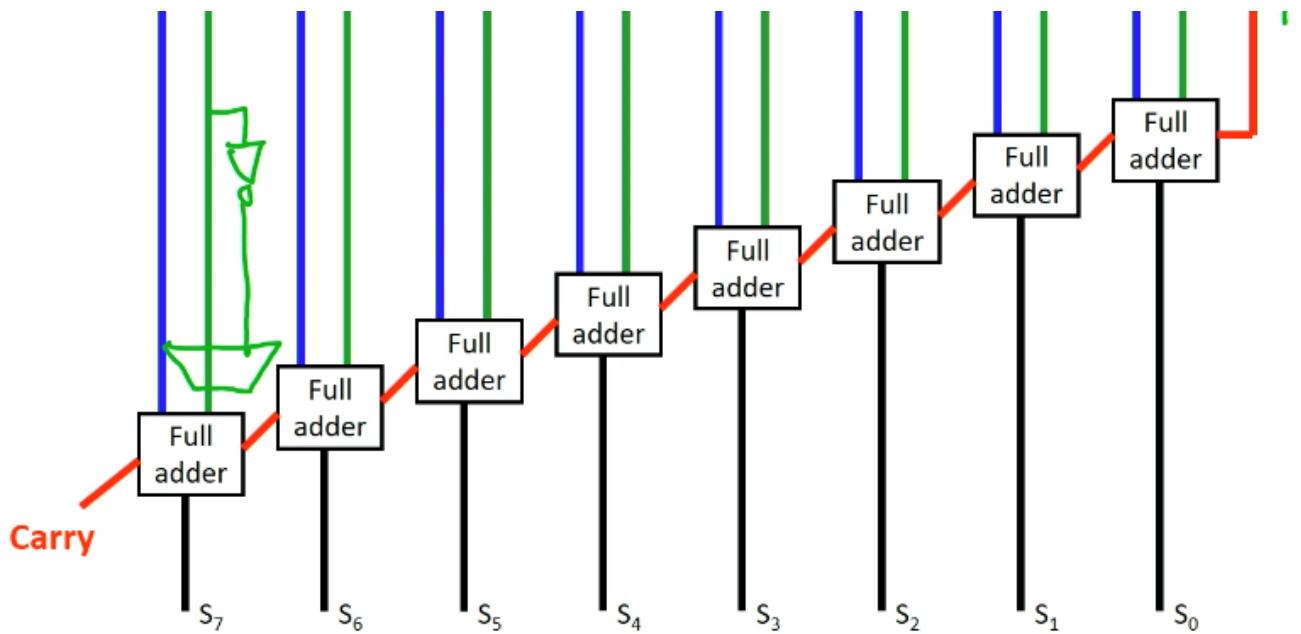
我们知道 $A - B = A + (-B)$,

并且 2'complement 下, $-B = \sim B + 1$

于是我们发现了一件巧妙的事情: 一般情况下, 我们把第 0 个 bit 的 input carry bit 设置为 0; 而我们只需要把这个第 0 位 carry bit 设置为 1, 然后 invert B, 就可以得到 $A - B$

我们可以在 B 上加一个 mux, 这样就可以通过控制是否 flip B 来控制这个 adder 和 subtracter 之间的转化, 让其多功能。





This will be very slow for 32 or 64 bit adds, but is sufficient for our needs

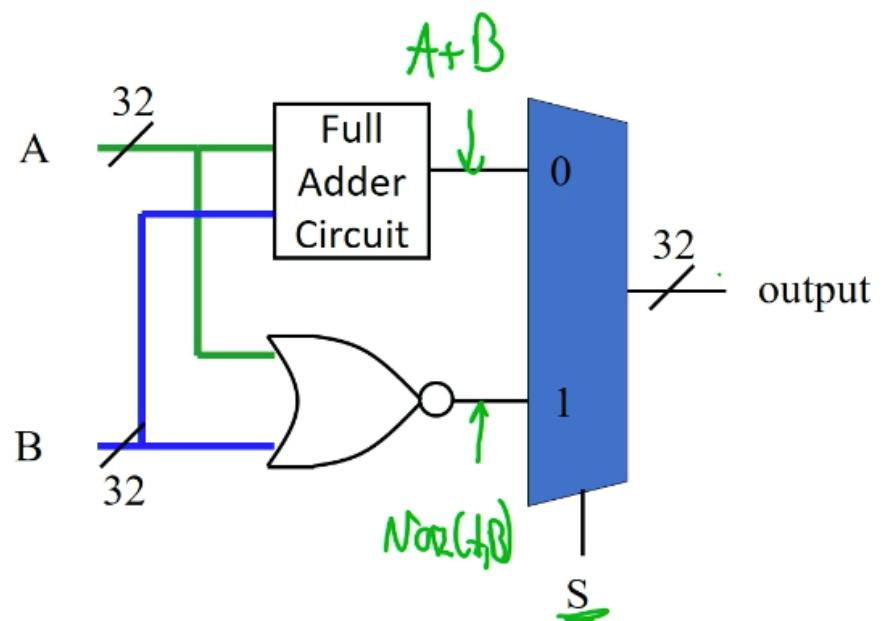
ALU design

我们需要一个计算单元把 full adder 和 nor 结合起来，这样它就可以用来做到任何的基础运算。

ALU 就是 Arithmetic Logic Unit。我们很自然想到把一个 32-bit full adder 和一个 32 位 input 的复合 nor gate 的输出用一个 mux 连接起来就可以实现一个 ALU.

(这只是对于 LC2K，其他更复杂的 processor 融合了更多函数)

- This is a basic ALU (Arithmetic Logic Unit)
- It is the heart of a computer processor!



当 $S = 1$ 时这是个 nor，当 $S = 0$ 时这是个 32-bit adder.

Propagation delay

当 gate 的 input 变化时，output 不是立刻变化的。gate 的处理需要一定时间。因为 speed of light 也是有限的，transmit through wires 需要一定时间；并且电压变化也不是一瞬间的，电压变化到一定值使得 transistor switch 也需要一定时间。

所以有了 Propagation delay. 也就是对于每个逻辑门（可以忽略 wires 的电速传输导致的 delay），我们都要考虑它接受 Input 到输出 output 之间，逻辑门处理的 delay.

overall delay 就是 **delay** 最大的一个串联路线的 **delay**.

我们需要考虑 delay，因为我们需要保证我们留下了充足的时间让它 propagate，以至于我们得到的值确实是逻辑结果而不是误差导致的中间结果

Lec 9: Sequential Logic

Combinatorial Logic 指的就是 circuits whose input 只取决于 current input. 即不关心时间，只需要 operands 就能产生结果

但是 combinatorial logic 并不能形成一个 computer. computer 会 remember previous inputs 并根据 history 对相同的 inputs 有不同的表现。

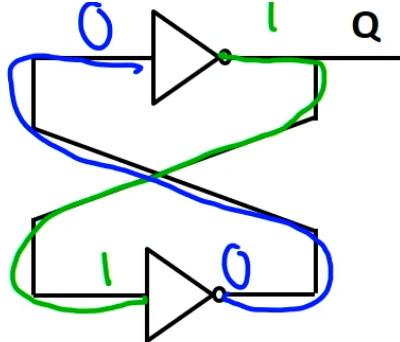
具体的理解是：

我们的 program 有 state. states 就是当前程序的状态：比如各个变量的值，PC 在哪一条指令上，regs 的内容以及 memory 上的内容等等。

而 input 则是外部的数据输入：比如 cin 等等

对于不同的指令，以及变量值得不同，计算机处理相同的输入内容的行为不同。

我们可以用 gates 来建立 seq logic. 关键在于 feedback: 一个 gate 的 output 会 feed back into its own inputs.



一个这样的 cell 会记住初始值，一直停留在初始值上。但是我们无法再改变它。所以我们需要一些输入来控制一个单元，让它能够变成这个 cell，但是也可以控制它变化。

SR latch

latch 即一个触发器

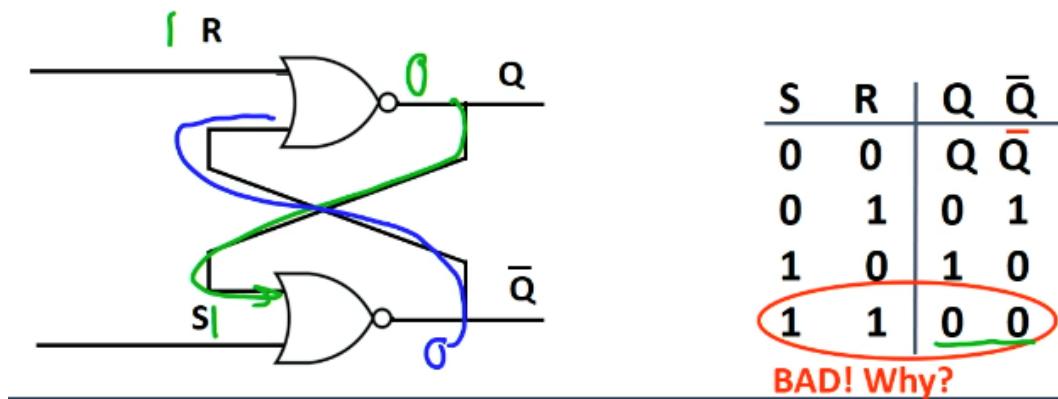
SR 用来 implement memory，表示要不要把一个 bit 设成 0/1

S, R 是两个 bit input。S 代表 set, R 代表 reset

$S = 0, R = 0$: 什么都不做，Q 保持原先状态（本来是1就是1，本来是0就是0），因为这个时候被转化成了这个稳定的

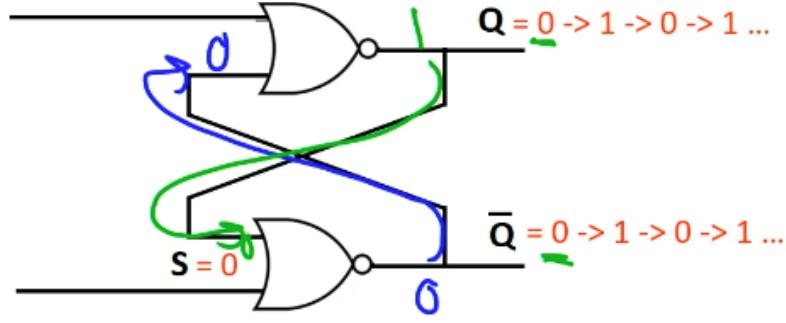
$S = 1, R = 0$ (set): 输出 $Q = 1$ ($Q \text{ reverse} = 0$)

$S = 0, R = 1$ (reset): 输出 $Q = 0$ ($Q \text{ reverse} = 1$)



$S, R = 1$: undefined behavior. 这个时候输出 Q 和 Q^\wedge 都变成了 0，暂时稳定，但是问题是：如果我们这个时候调整 S, R 都变成 0，那么 Q 和 Q^\wedge 都会不停 rapidly osilating between 0 and 1，circuit unstable 且不可预测。

$\text{R} = 0$

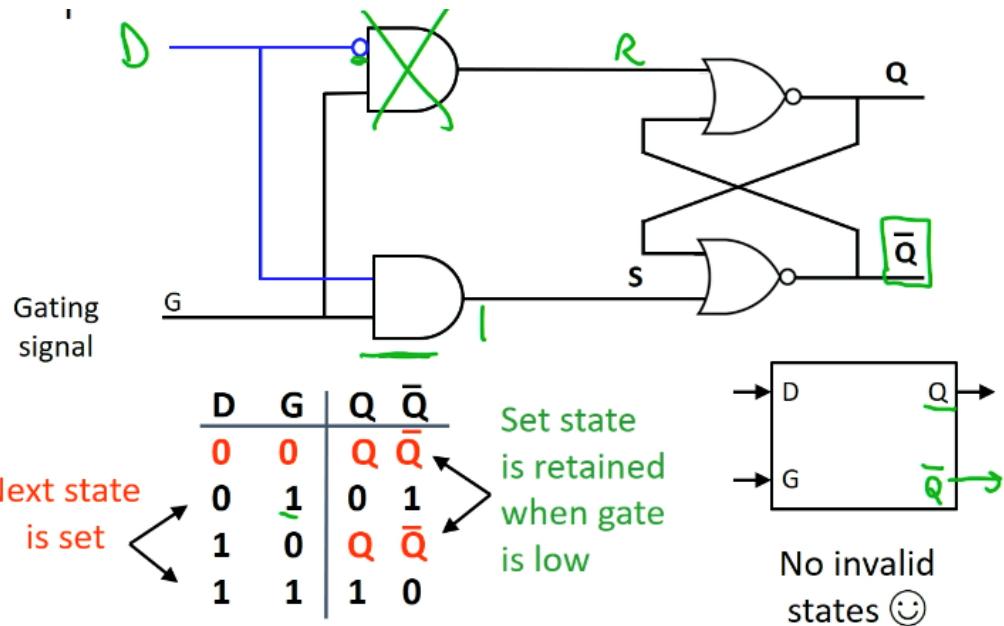


D latch (improved SR latch)

D 表示 Data

G 表示 Gate.

这个改进版本的好处就是不会再出现 undefined behavior, 保证了数据安全。

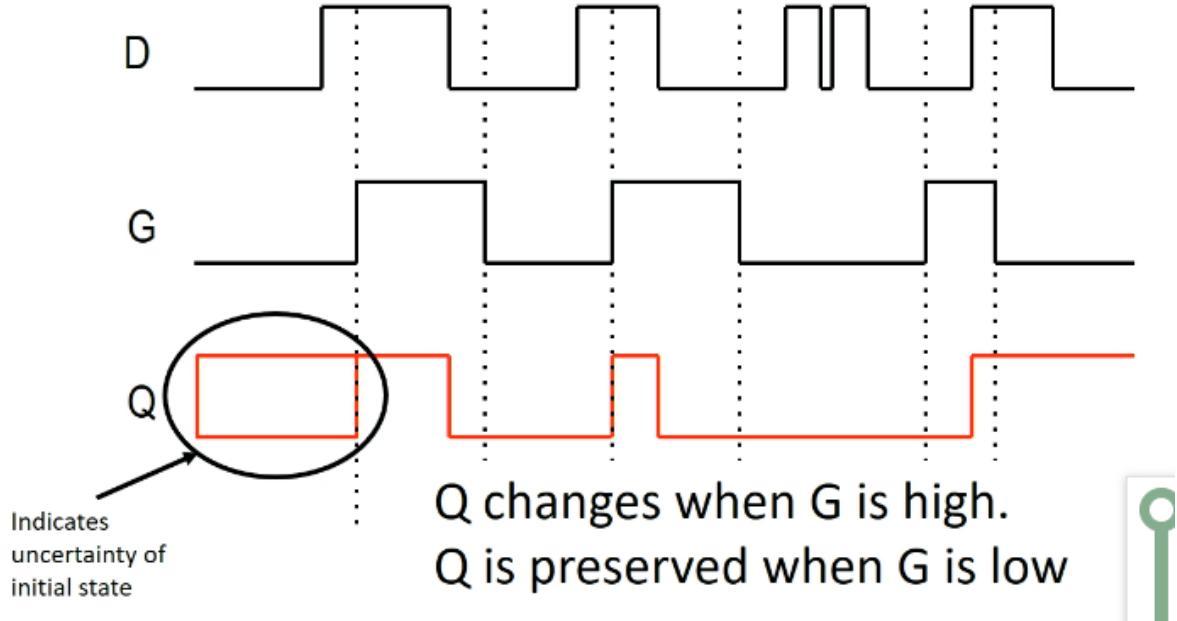


现在设置 $G = 1$: 允许改写 data, $Q = D$ 。我们称这个状态下 latch 是 transparent 的 (允许修改)

设置 $G = 0$: 不允许改写 data, 不论 D 是多少, Q 都保持原状

画 D-latch Timing Diagram

: : : : : :



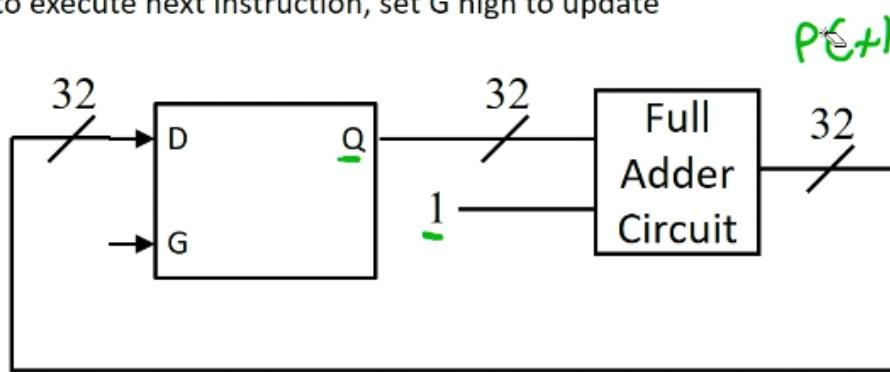
在刚开始的时间: G low, Q 保持之前的状态 (不知道), 所以画两条线表示 Uncertainty.

之后, 我们的原则: 只有 G 开时改写, G 关上时保持状态。

D flip flop (improved D latch)

D latch 的 problem: G 必须被 set very precisely

- Can we use D-latches to build our PC logic?
- Idea:
 - Use 32 latches to hold current PC, send output Q to memory
 - Also pass output Q into 32-bit adder to increment by 1 (for word-addressable system)
 - Wrap sum around back into D as "next PC"
 - Once ready to execute next instruction, set G high to update



我们可以尝试用 32 个 D latches 来储存 PC, 并且通过这样的循环, 每次当 ready to 执行下一个 instruction 的时候就

open gate 米控制 PLC++

但是问题是这个电路太过敏感。我们需要 PC 恰好被 ++ 一次 (或者特定数量)

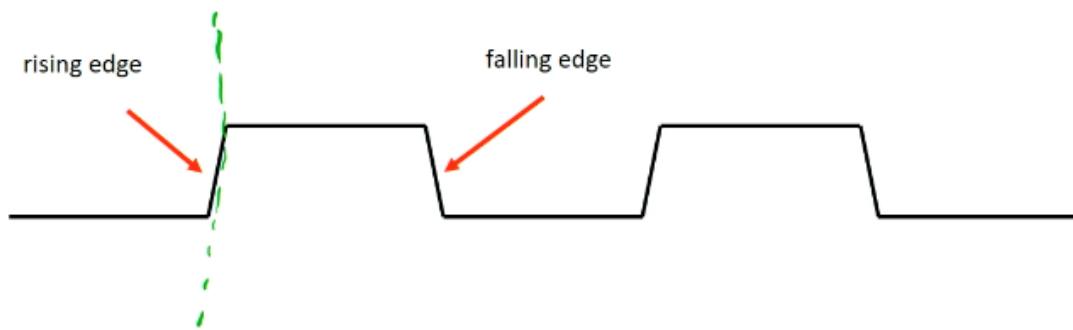
如果 G 设置的时间过长了那么信号可能会 propagate round 两次以至于 PC 被 increment twice; 如果 G 设置的时间过短可能 latch 没有 stabilize, 使得 PC 没来得及 increment

(FYI: set 一个 frequency 很高的 signal 是很难的)

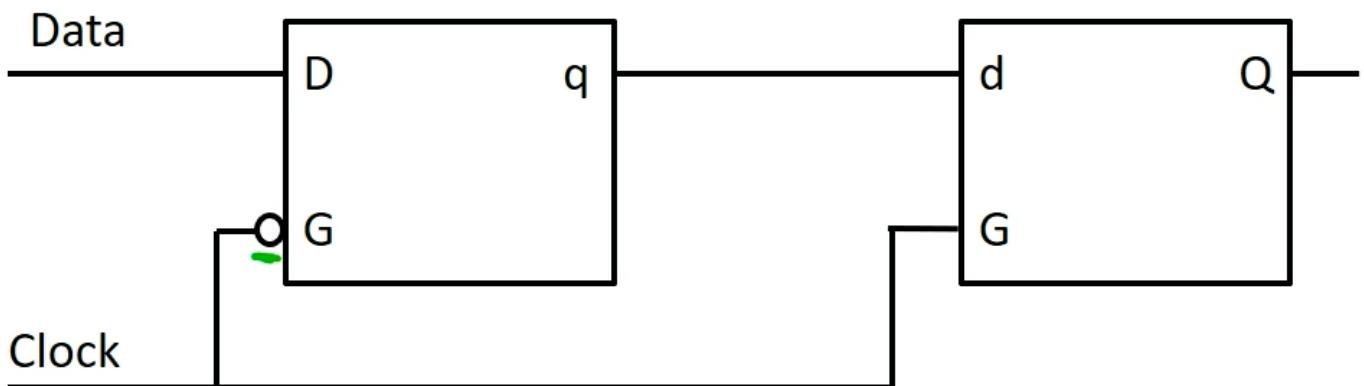
所以 timing 很难。

所以我们添加一个 clock: 一个在 0 和 1 之间 alternating 的固定 frequency 的 signal.

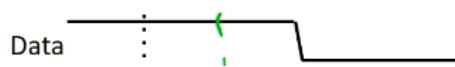
只在 clock 的 edge 上 write data.

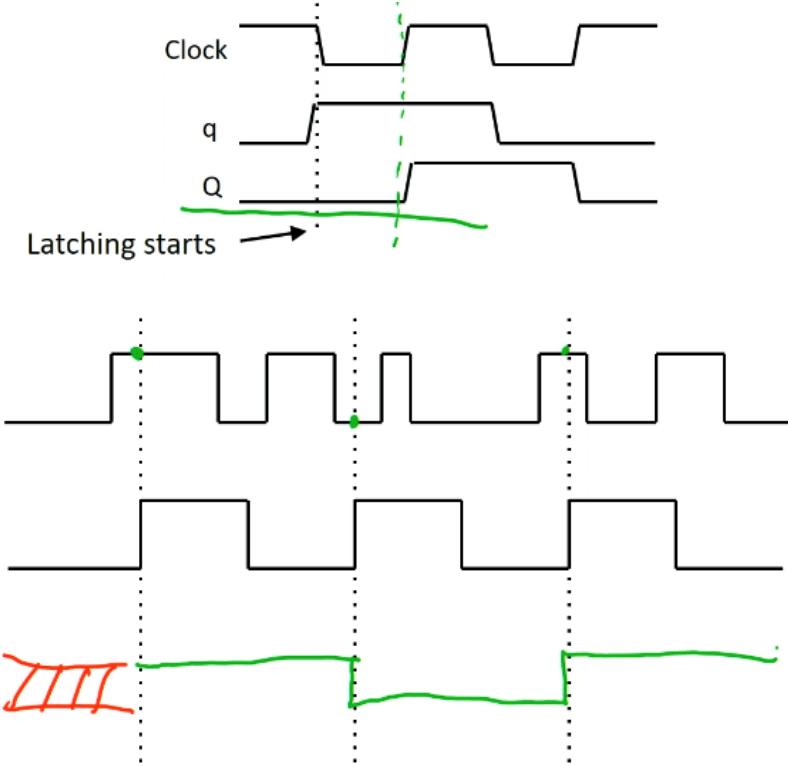


this class 只考虑在 rising edge 上写入 data 的固定的 Clock 接入方式. (Dually)



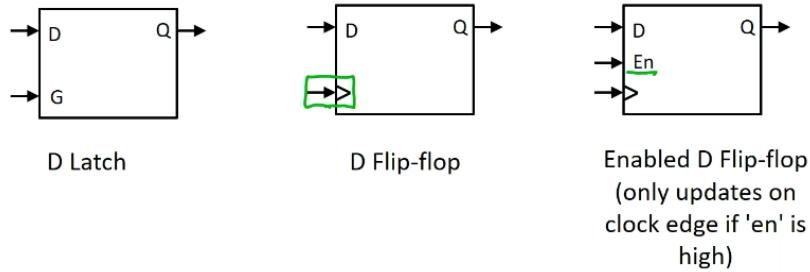
就是这样的接入方式。我们可以验证它只在 clock 从 0 变成 1 的 rising edge 上使得数据被写入 Q (可以但没必要





D flip flop 并不是无懈可击的，如果恰好，data 改写的时间就在 clock 的 rising edge 上，那就 bad. (reason: will know it if you take 270)

标识：普通 D latch, D flip-flop 和 Enabled D Flip-Flop （多加入 enable 信号，只有 enabled 时才会接收更改，



为什么 Flip Flop 可以避免 PC 被 increase 多次的问题

假设 PC 在一个时钟周期内需要递增一次，而在时钟高电平期间，递增信号传递到 D Latch。由于 D Latch 在高电平期间对输入开放，如果递增信号持续存在，PC 可能被递增多次，因为每次输入 D 变化时都会更新输出 Q。

当使用 D Flip-Flop 来存储 PC 时，即使 PC 的递增信号在时钟周期的高电平持续时间内保持有效，D Flip-Flop 只会在时钟的边沿时刻更新 PC 的值。因此，即使递增信号在时钟高电平期间存在很长时间，PC 也只会被递增一次。

(my question: 如果同一个 pc 递增信号经历了两个 clock edge 怎么办？如果一个 pc 递增信号在 clock edge 来到之前就消失怎么办？

答：不考虑。假设一个信号一定正好经历一次 clock edge.

Lec 10: Finite State Machine

combinatorial logic: 用来 implement 布尔表达式

sequential logic: 用来存储状态

现在我们学习如何

一个 FSM 的组成是

1. 有限个状态
2. N 个 inputs
3. M 个 outputs
4. Transition function $T(S, I) : states \times inputs \rightarrow states$, 把每个状态下的每个 input 都映射到一个新的状态
5. Output function: 分为两种: 如果只取决于 State, 那么就是 Moore Machine; 如果取决于 state 和 input, 那么就是 Mealy Machine

计算机中的一个state 就是 memory, reg files 和 regs 的值. 我们使用 Read Only Memory 来 implement (对于一个 ISA的) transition function: 即操作指令如何和 memory 进行交互

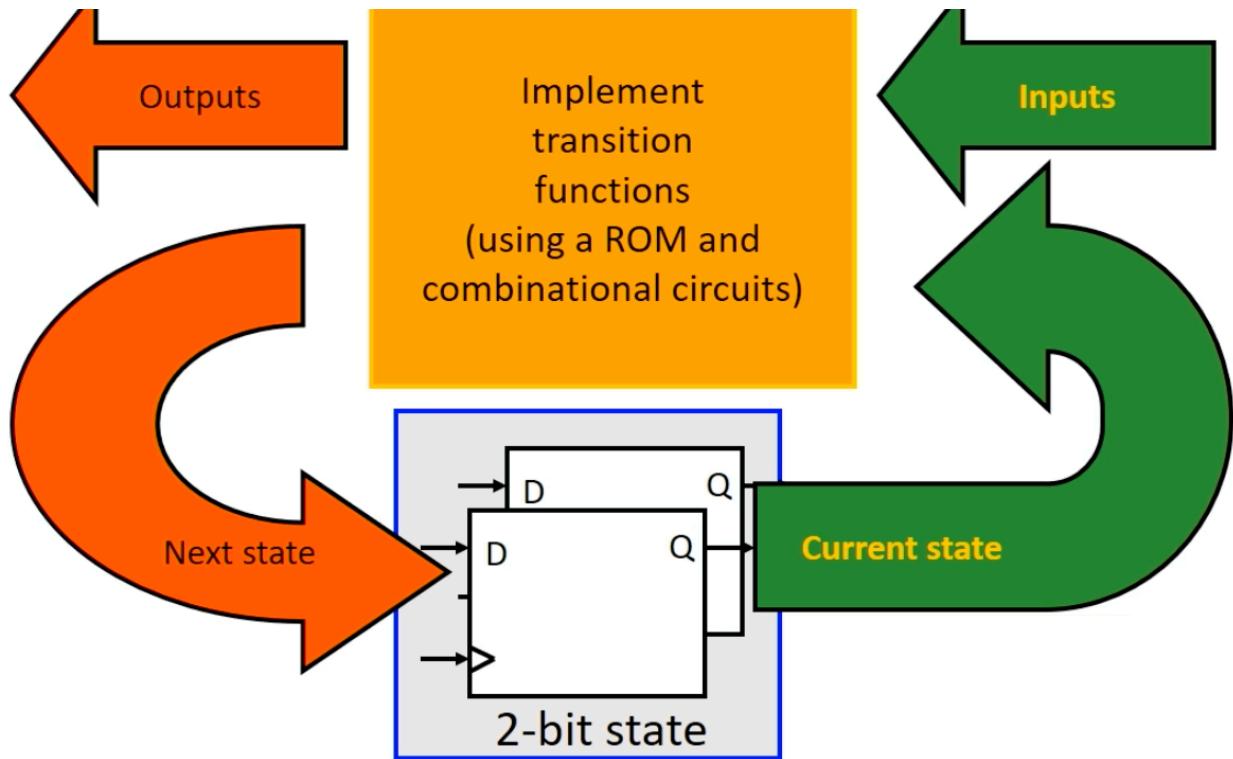
和 376 的 FSA 基本一样。区别是: FSA 只在 final state 产生 output 而 FSM 一直在产生 output; FSA will eventually stop, 但是 FSM 会有不停的 (ideally) input, 不会 stop.

FSM 就是一个每时每刻都接受 input, 并马上通过 input 来到 next state 并产生 output 的机器。

ROM(read only memory)

我们使用 programmable read only memory (每个 bit 只能修改第一次) 针对我们的 ISA 来 implement 出它的 FSM (即读写数据以及计算数据的 data path)

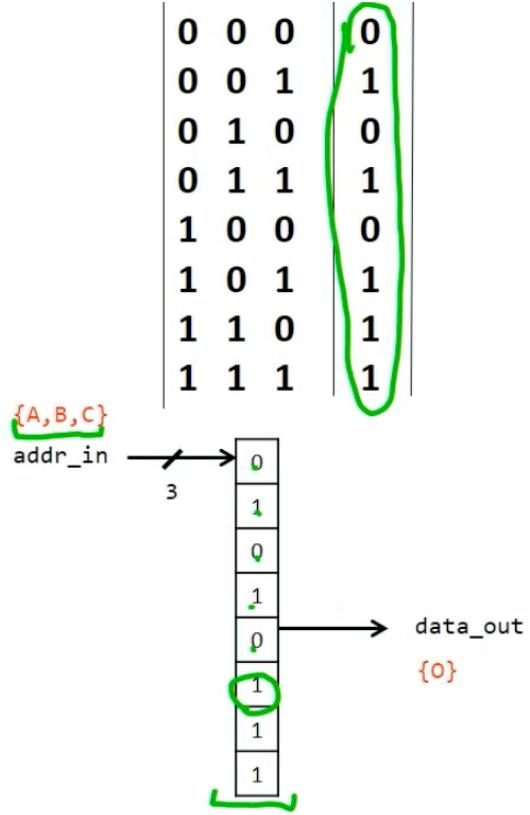




Idea: 由于有很多变量和输出结果不同，我们用 logic gates 来实现真值表太过于耗时间。

所以我们考虑直接把整个真值表储存进某个 Read Only memory 里。可以直接买一个 read only memory bar 来实现。

A	B	C	<u>Q</u>
---	---	---	----------



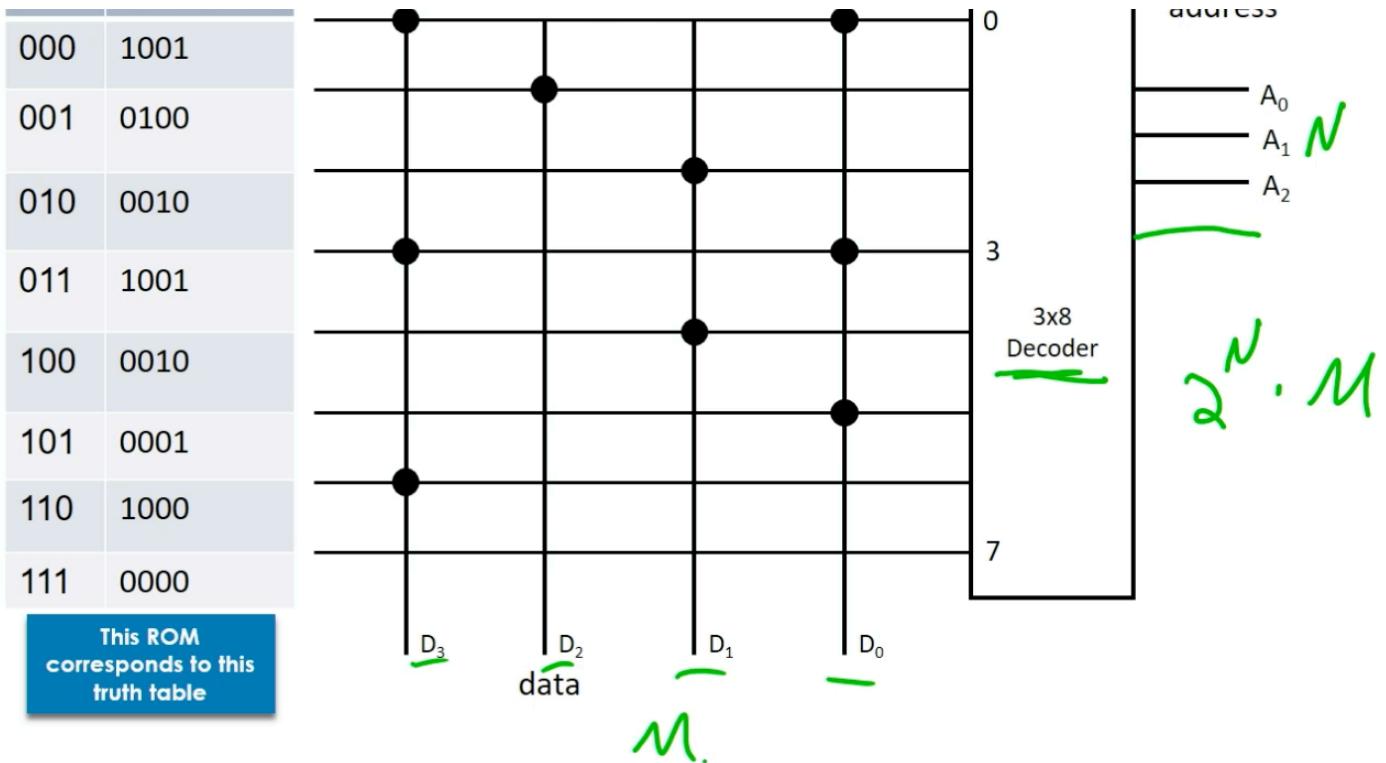
ROM 的使用是这样的：

这是一个 8 entry 4 bit 的 ROM.

我们输入一个 3 位的 bits 用 3×8 decoder 把它的输出转化为 8 位只有一位是 1 的 bits, 于是每一个输入就对应了 decoder 的一个 horizontal line.

我们可以通过调整 data output 和 horizontal line 交线上的 diodes 来编辑 ROM (只能编一次, 因为调整就是把它烧掉), 控制这一条 horizontal line 对应的输出。

这就实现了每个 state 的不同输出



这种存储 state output 的方式很常见，也有缺点：如果我们要多加入一个 bit 的 input，我们就要 double the size of ROM. 所以储存大小的需求是 exponential 的

计算 size of ROM

Size of ROM = #of ROM entries * size of each entry.

其中 #size of ROM entries = $2^{\text{input size}}$, size of each entry = outputsize

所以

$$\text{sizeROM} = 2^{\text{input size}} * \text{outputsize} \quad (1)$$

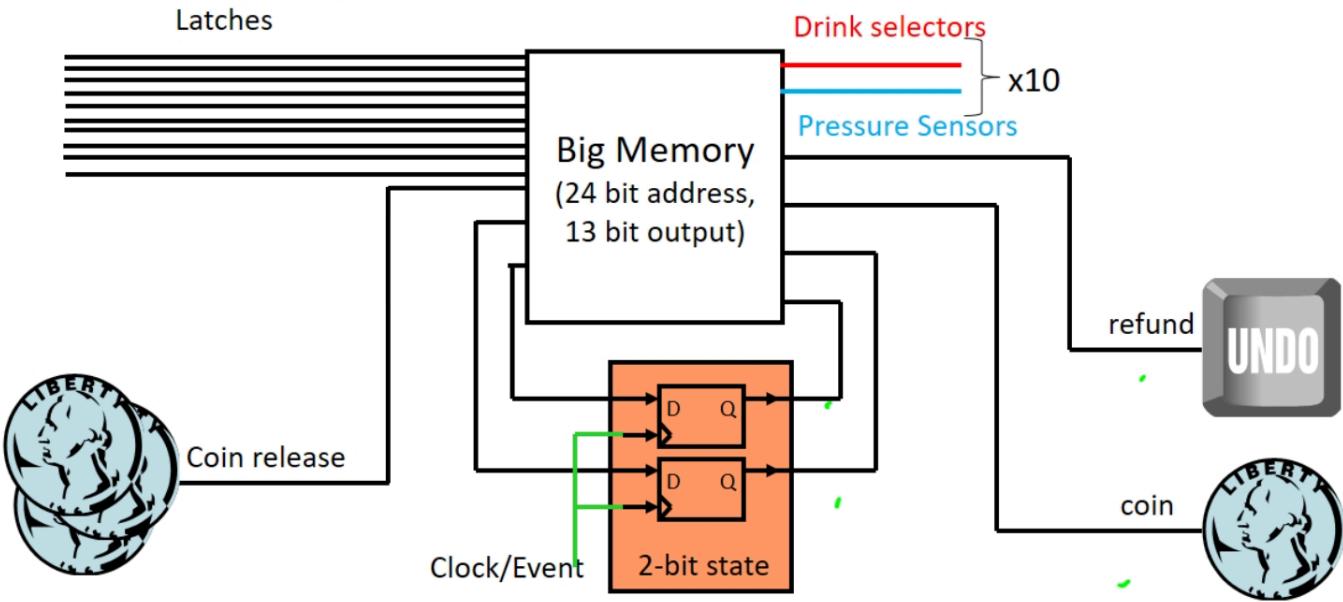
对于 24 位地址的输入，假设我们 ROM memory 宽度是 13 bit，那么我们需要 $2^{24} * 13 = 218,103,808 \text{ bits} = 26 \text{ MB ROM}$. 如果我们买六美元 4mb 的 ROM，我们需要大改 42 美元才可以实现

(并且我们需要编辑这些 ROM，显然不太可行

所以我们需要结合 combinatorial logic 和 ROM 来实现 transition 来优化我们需要的 ROM 大小

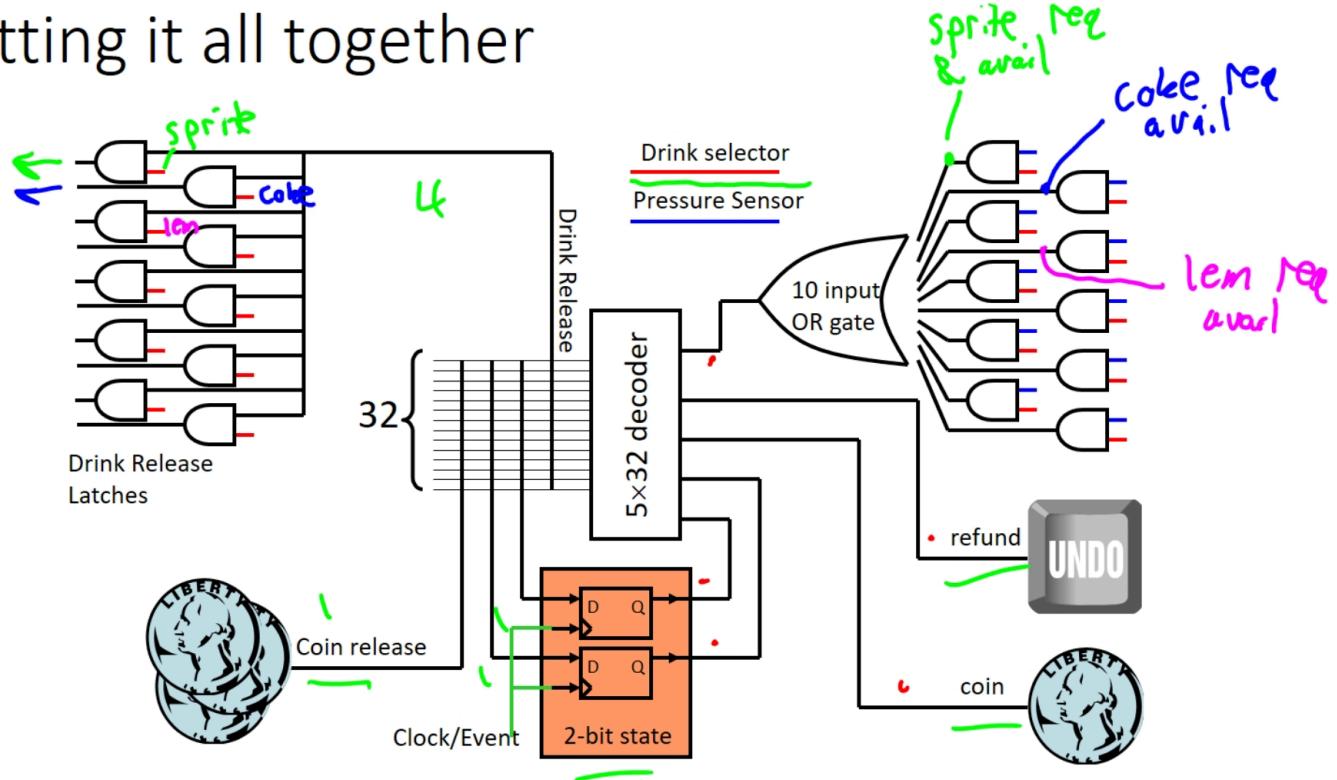
ex:

优化前：ROM = $2^{24} * 13 \text{ bits}$



优化后: ROM = $2^5 * 4$ bits

Putting it all together



Lec 11: Single-Cycle Datapath

我们需要设计一个 General purpose processor.

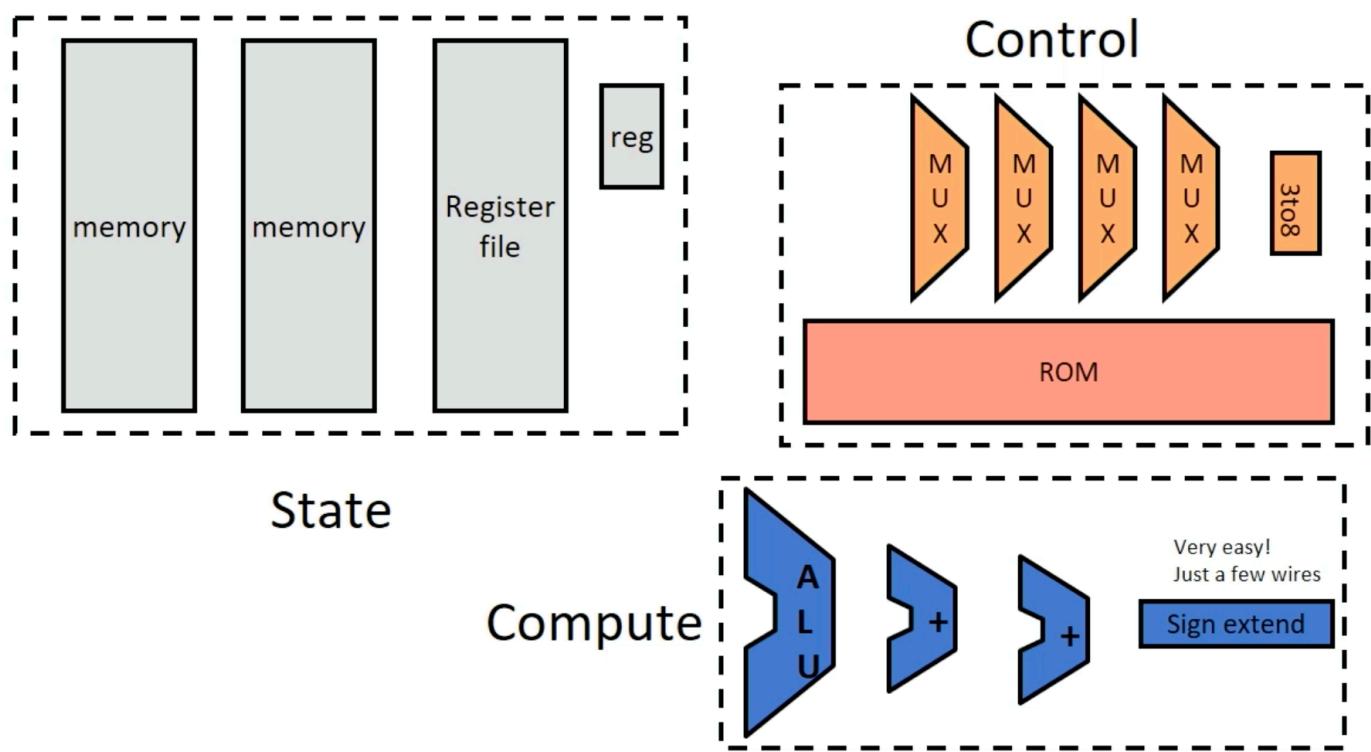
它需要做的：

1. fetch instructions
2. decode instructions (即把 instructions 输入给 control ROM)
3. 通过 ROM 来控制 data movement

包括 pc++, reading regs, ALU control 等

LC2K 中，ROM 接受一个 3x8 decoder，把 3 位的 opcode decode 为 00000001 - 10000000，对应八个不同的八位数，放在合适的地方（比如作为某个 mux 的选择 bit 等）来实现不同的操作

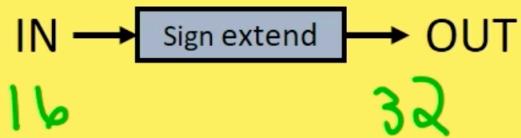
4. 用 clock 来 drive all



Note: 我们假设任何 memory 都是一个 array of D flip flops, 实际情况更复杂，但是我们这样假设

Note2: sign extension 运算

Sign Extension Unit



Sign extend (SE) input by replicating the MSB to width of output

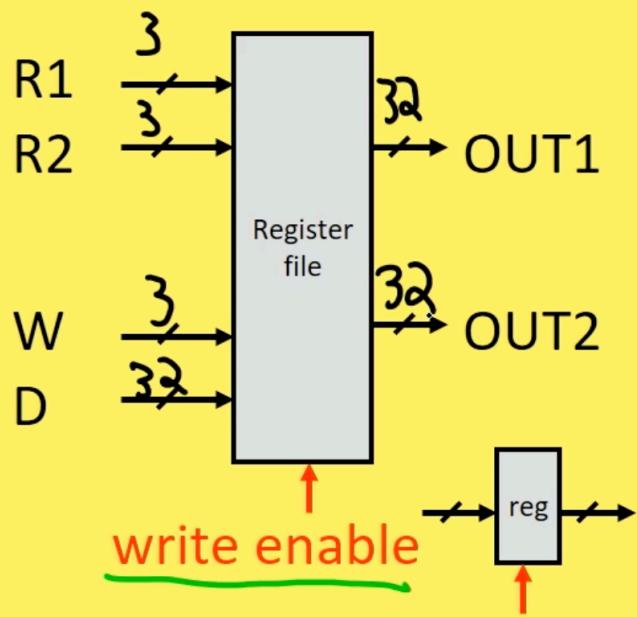
$$OUT(31:0) = SE(IN(15:0))$$

$$OUT(31:16) = IN(15)$$

$$OUT(15:0) = IN(15:0)$$

State Building Blocks: reg files, memory

Register File or Register



Small/fast memory to store temporary values

n entries (LC2 = 8)

r read ports (LC2 = 2)

w write ports (LC2 = 1)

* Ri specifies register number to read

* W specifies register number to write

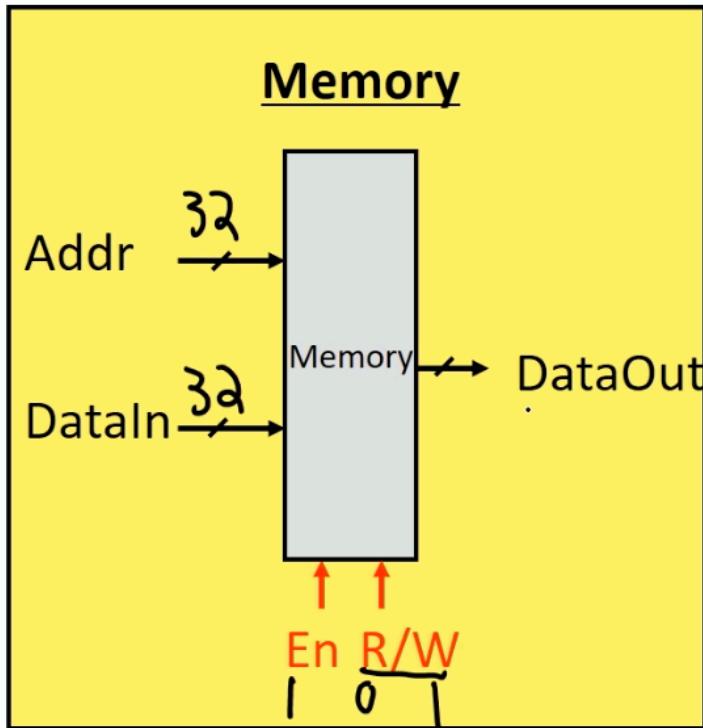
* D specifies data to write

Reg file 就是所有 regs 的集合。

我们使用这样的一个 reg file memory block 来储存 memory，通过逻辑门取其中的第 R1 个和第 R2 个以及第 D 个：R1, R2 表示读取的两个 regs, D 表示要写入的 reg

W 表示要写入的数据

还有组合进它的逻辑的是一个 enable bit, 用 mux 来控制是否要打开 write. 不需要改写 reg value 的指令可以不 enable 它, 使得速度更快



Slower storage structure
to hold large amounts of
stuff.

Use 2 memories for LC2K
* Instructions
* Data
* 65,536 total words

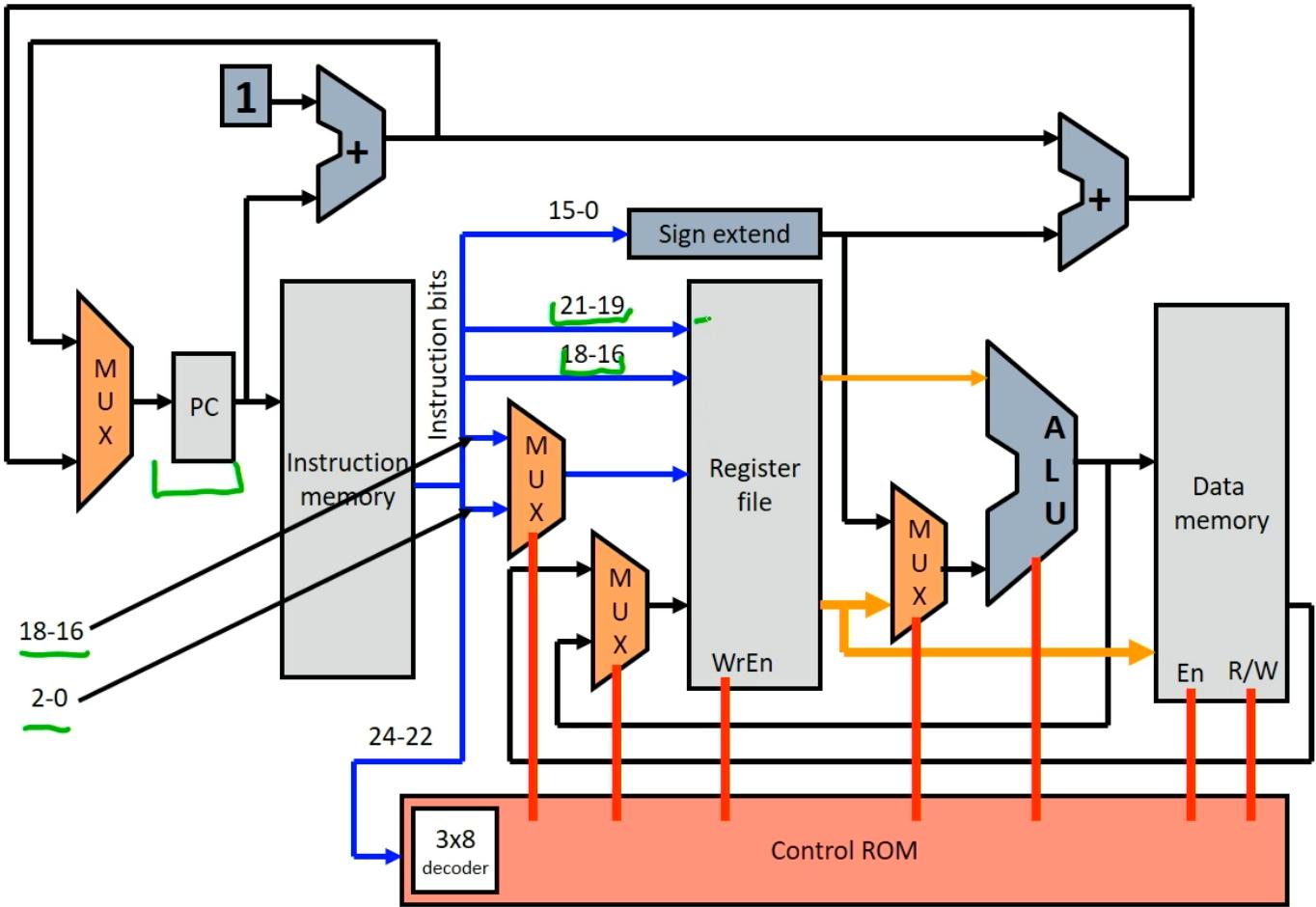
data memory: 更慢的 memory。这是因为我们总是先处理 reg file, 然后 reg file 中输出的 reg values 再才可能输入到 data memory 里, 用以交互, 储存 regs 里放不下的值

同样支持读写, 需要一个 enable bit 和一个 mux 控制 enable

现在我们做一个 single-cycle datapath for LC2K: 任意 instruction 都在一个 clock cycle 内完成

Overall View

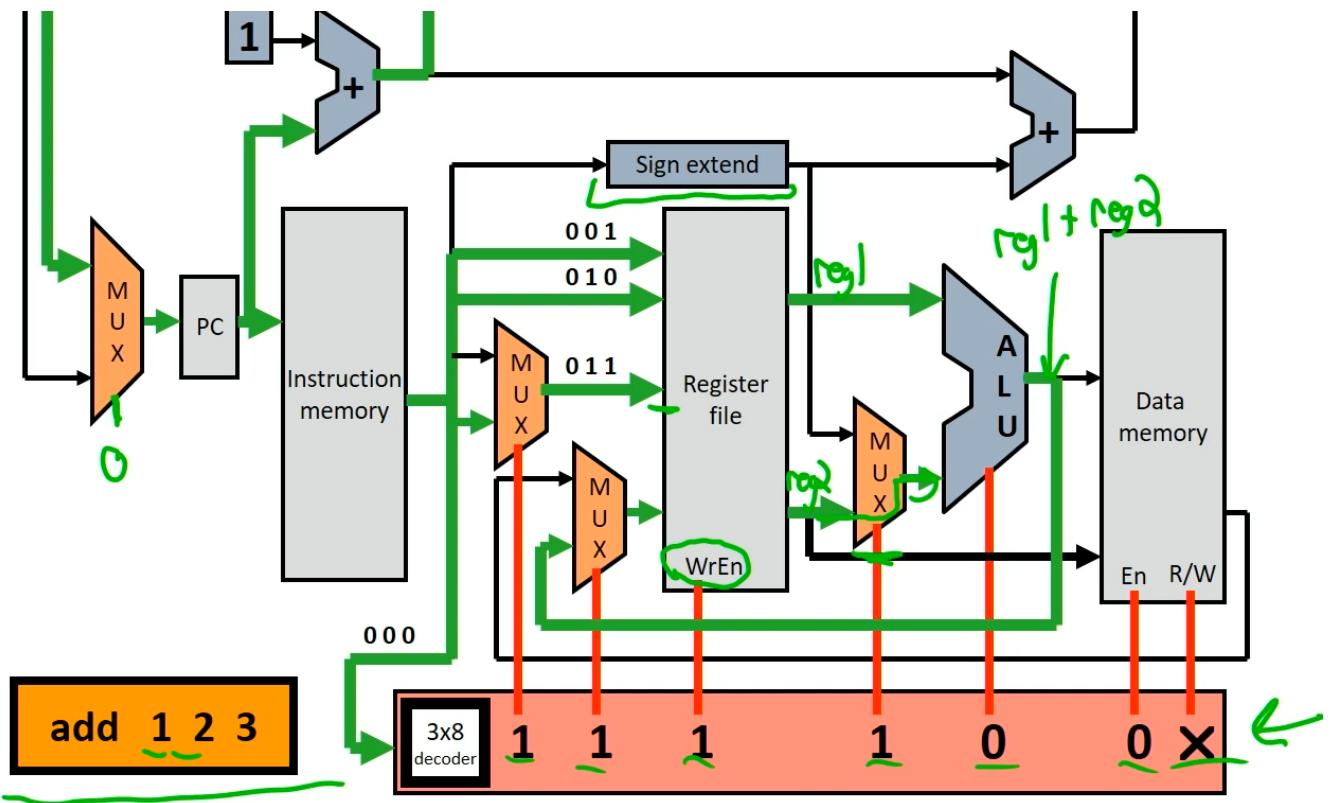
LC2K Datapath Implementation



ADD/NOR

add regA, regB, destR

即: dest R = regA + regB; PC++



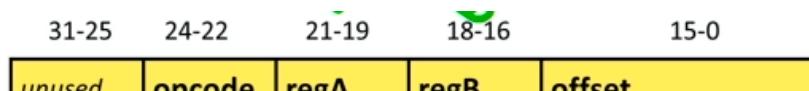
opcode: 000, decode: 00000000

1. 第 6 个 ROM bit: mux 设置为 1, 因为 regA 是 bit 2-0
2. 第 4 个 ROM bit: reg write enable 设置为 1, 因为要改写 destR
3. 第 3 个 ROM bit: mux 设置为 1, 因为我们需要获取 regfile 的第二个 output 而不是上面 sign extend 的 offsetfield 的值
4. 第 2 个 ROM bit: ALU 设置为 0, 因为我们需要 add 而不是 nor
5. 第 5 个 ROM bit: MUX 设置为 1, 因为我们要的不是 data memory write 回 reg files, 而是要这个时候我们的 reg1+reg2 的结果 write 回 destR.
6. 第 1 个 ROM bit: enable 设置为0, 因为不需要写入 data memory.
7. 第 0 个 ROM bit: 随便, 因为已经设置 enable 0 了, 这个时候 R/W 都不生效

用时: 读指令 (access memory) + read reg + ALU + write reg

nor: 除了第 2 个 ROM bit 设置为 1, 获取 Nor 结果, 其他都一样。

LW/SW





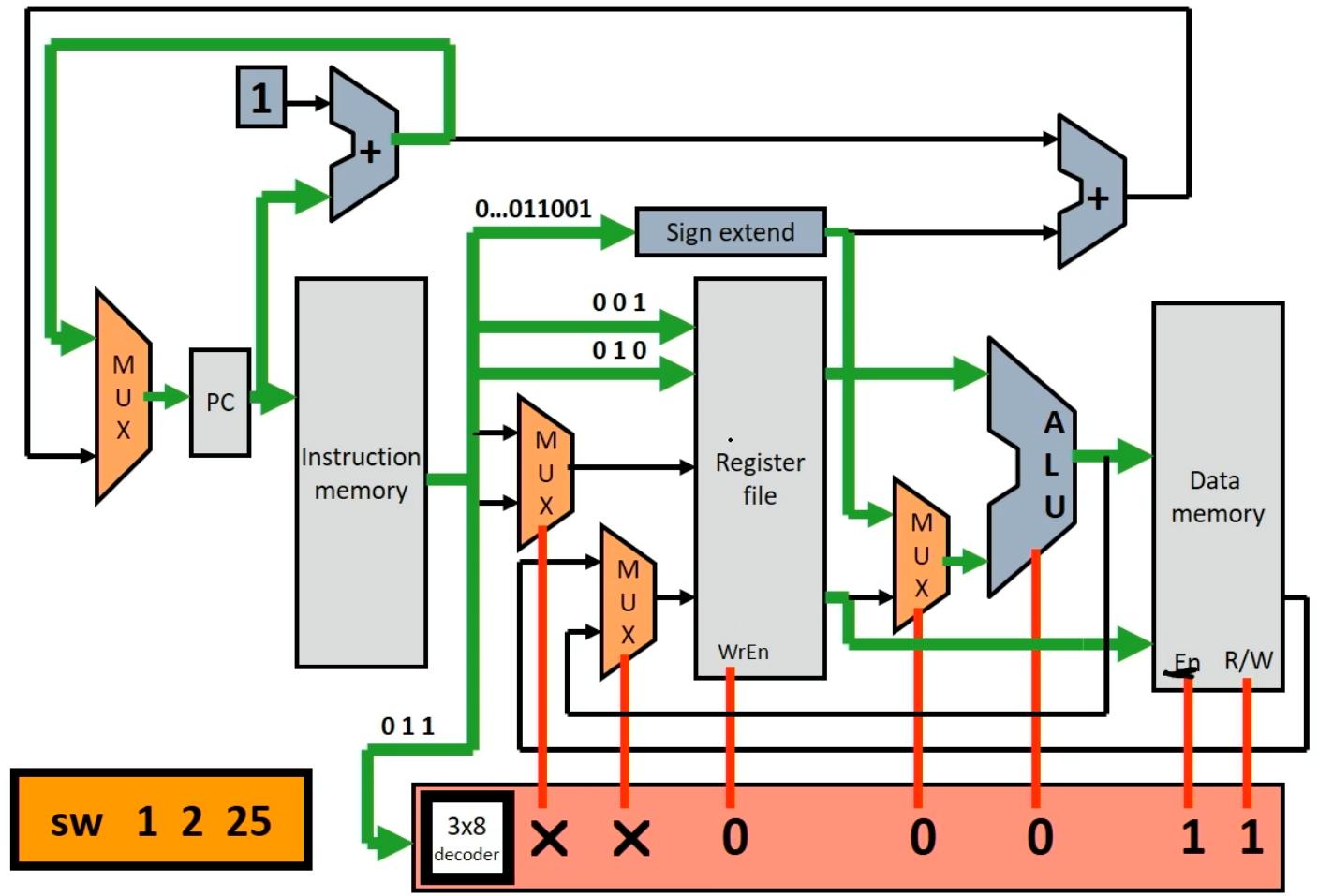
lw regA, regB, offset, 即:

$$\text{regB} = M[\text{regA} + \text{offset}]; \text{PC}++$$

write enabled. reg1 在 ALU 中和第三个 mux 里来的 sign extend offset 进行相加, 结果进入 data 作为 read memory 的位置, 把 read 出来的 memory 又传输给了第二个 mux, 于是第一个 mux 代表的 reg 的值被改写为第二个 mux 的 output 的值

这里 R/W 的 0 表示 R, read.

sw: 即 $M[\text{regA} + \text{offset}] = \text{regB}; \text{PC}++$



耗时

sw: get inst(read mem) + read reg + ALU + read mem

lw: get inst(read mem) + read reg + ALU + read mem + write reg

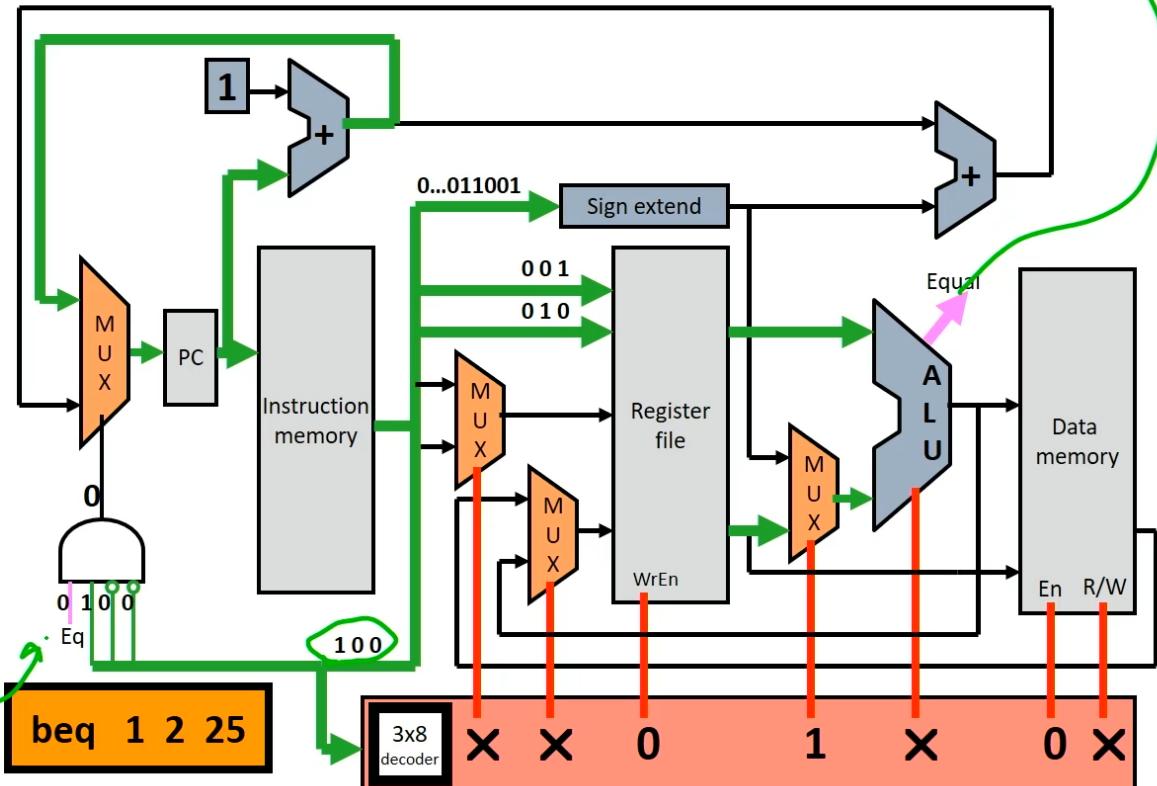
BEQ

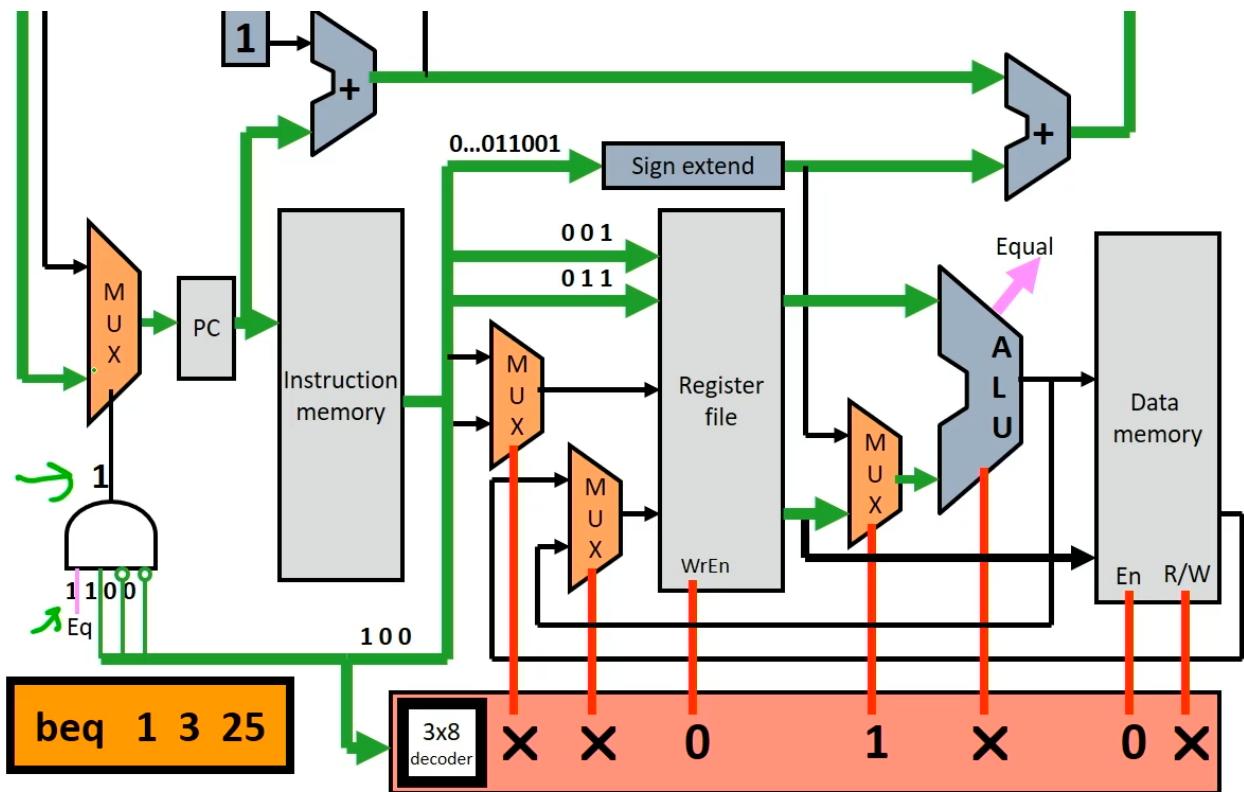
beq regA, regB, offset

if (regA == regB) 则 PC += 1+offset

else PC++

Executing “not taken” BEQ Instruction on LC2K Datapath





对于 `beq`, 我们需要另一个额外的通路。设置一个四 bits 的 and, `b3` 获取 `ALU` 的结果看是否是1, `b[2:0]` 判断 `opcode` 是否是 `beq(100)`.

我们在 `ALU` 中判断 `regA` 是否等于 `regB` 的方法即: $\text{not}(\text{XOR}(A,B)) = (A \text{ nor } (A \text{ nor } B)) \text{ or } (B \text{ nor } (A \text{ nor } B))$

耗时

`read inst(mem) + read reg + ALU`

JALR

`jalr`: jump and link register

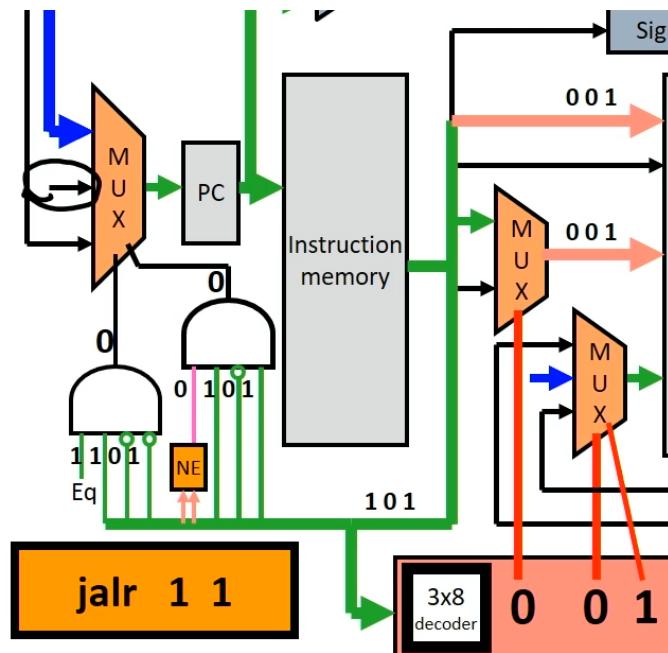
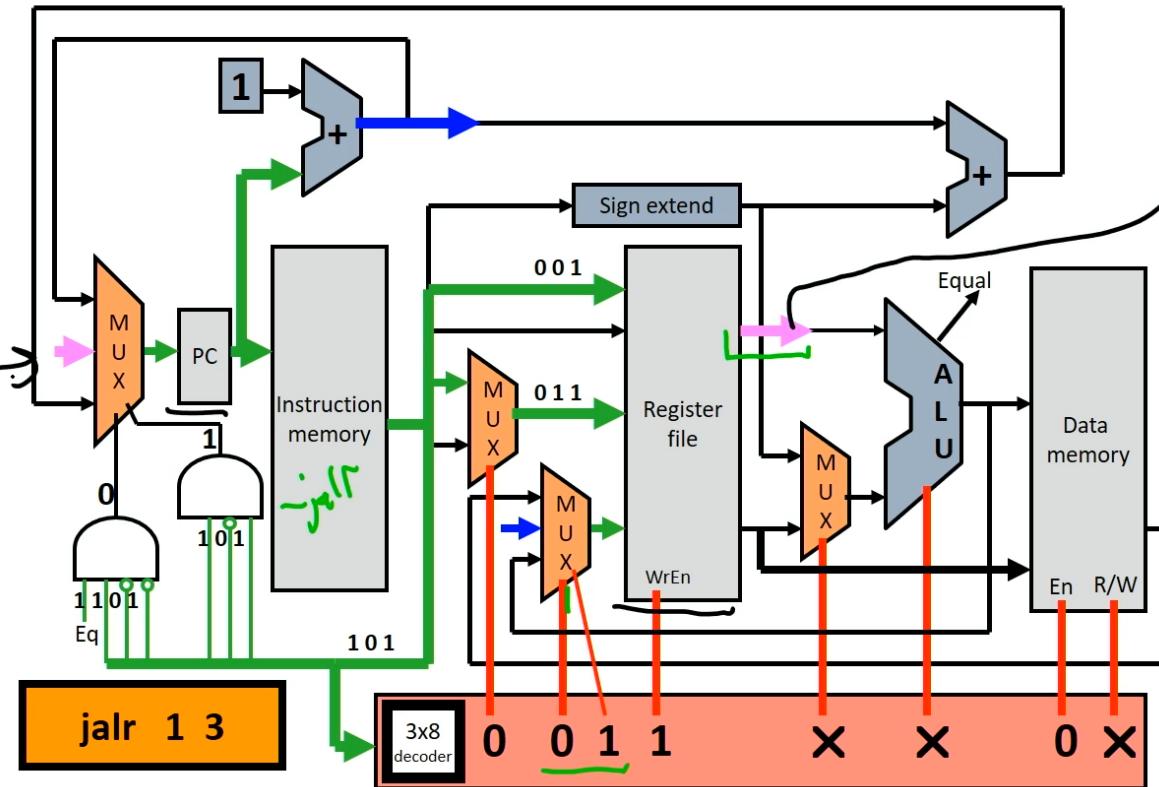
即:

`write PC+1 into regB`

`regA = PC`

这是一个 Ugly instruction。。

Executing a JALR Instruction



jalr 也需要额外的通路。

1. pc+1 的结果要通到第二个 mux (图中蓝色) , 并且要新加一个 selection bit, 选择 01 作为二位的 selection bits (pc+1 通到中间)
2. register file 的输出要通到第0个 mux (图中粉色) , 也是一样需要两个 control bits (因为一共有三个 inputs, 其实还有隐形的第四个但总是0)

这两个 control Bits = 01 (正确的 control bit) 当且仅当 opcode = 101 (jalr) 并且 regA 不等于 reg B。我们需要加一个判断并通到第二个 control bit 的控制 AND 门的最高位上，用一个 not equal 的判断逻辑门。

这是因为：我们难以处理 regA = regB 的状态。这个时候我们做的事情等于存 Pc 以及 pc++.

但是问题是：我们 update PC 和 reg file 是在同一时刻的。会导致我们把旧的 PC (没有++) 更新到 PC 上。所以我们需要判断 regA 不等于 regB。

Halt: 更加复杂。我们实际上并不能真的停止运行，只能 transfer control over 其他的 running programs. 这里不 implement.

Lec 12 Multi-Cycle Datapath & Pipelining

对于 single-cycle 而言，所有 Instructions 都 run at the speed of the slowest instruction. (最慢的指令决定 clock 周期，这是为了统一 clock 的周期，让最慢的指令也可以在一个 clock 内运行完)

如果我们想添加一个 long instruction，那么整体性能将极大下降；即便我们可以优化很多模块，也完全没用。

并且我们并不能 reuse processor 的任何 part

比如如果最长的指令是 lw, 8ns

有一百条指令，single cycle 运行延迟就是 800ns

我们希望每个指令有各自的运行延迟

Multi-Cycle Execution Overview

每个 instruction 都 take multiple cycles

cycle time reduced

Slower instructions take more cycles, faster instructions take fewer cycles

这样我们就可以通过优化一个操作来优化一个 clock cycle，从而让全部运行时间都降低，而不受短板的限制。并且我们每个 cycle 可以 reuse datapath.

为了完成这个优化，我们需要：

1. 更多，更 wider 的 MUXes, 为了 **reuse elements for different purposes.**
2. 更多 regs 用来记住同一个指令的上一个 cycle 的 output
3. 更复杂的 control

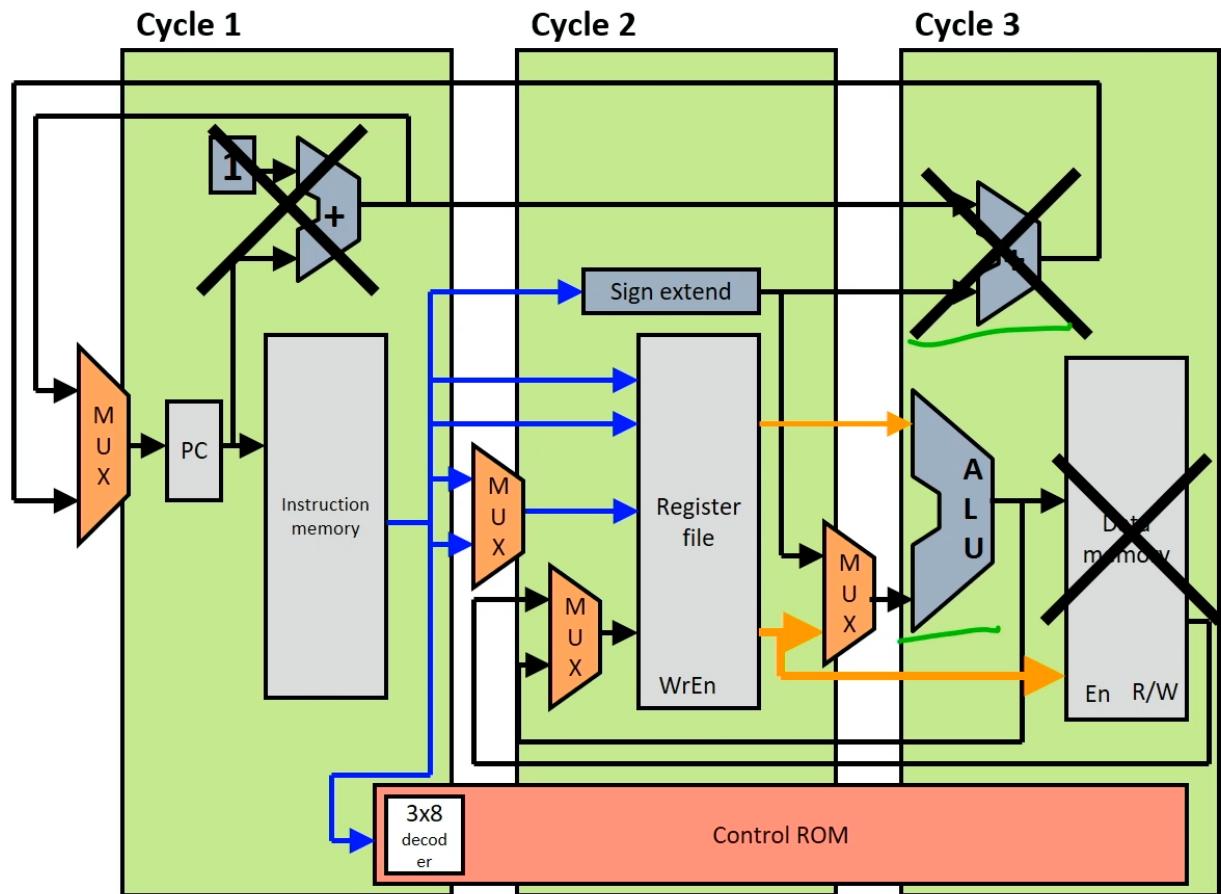
做法有很多种，我们可以选择：把一个 instruction 分成好几个 discrete 阶段

cycle 1: fetch instruction from memory

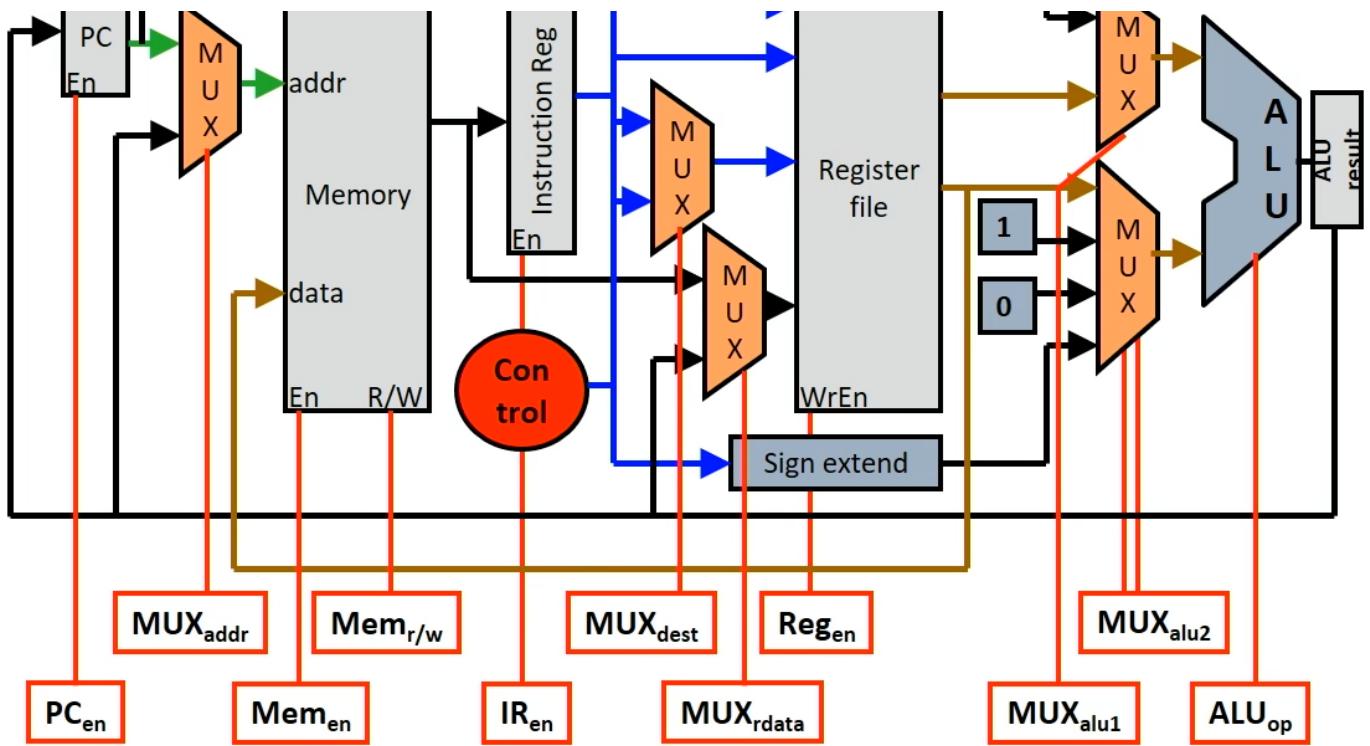
cycle 2: decode instruction

cycle 3+: 执行 instruction

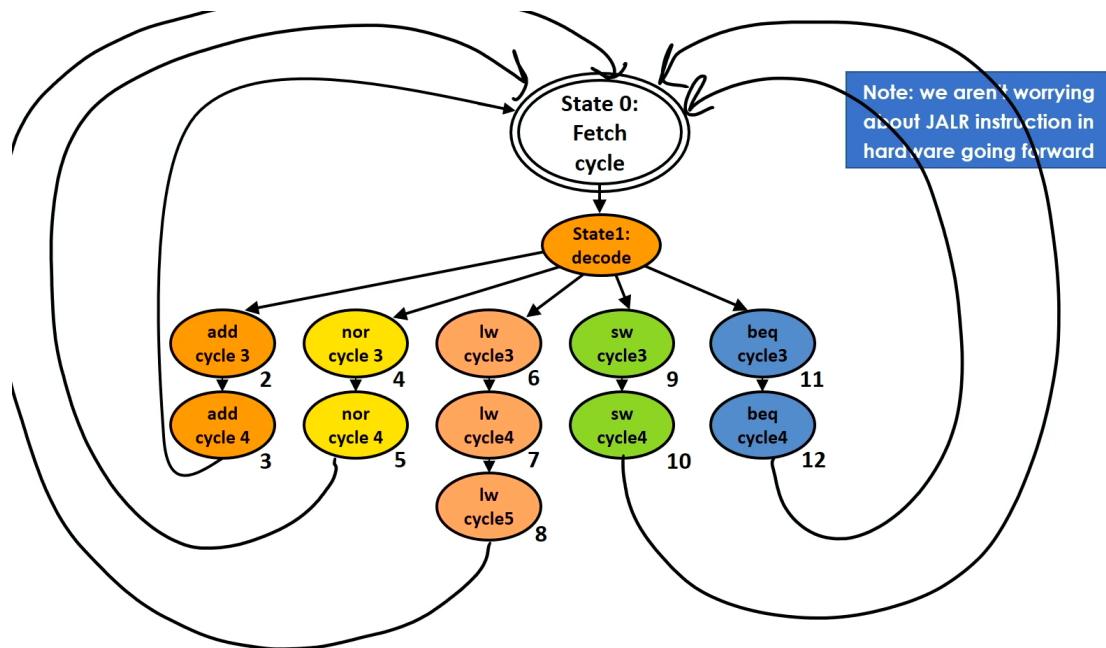
LC2K Datapath – cycle groups



Multicycle datapath 的 idea 就是：用更多的 control 换取单次 cycle 更短的时间，通过多次的 cycle 来实现一个指令



我们发现：ROM 的宽度，即一个 state 的 output，应当有 12 个 bits，如上图（MUXalu2 是 4-mux，有两个 bits）

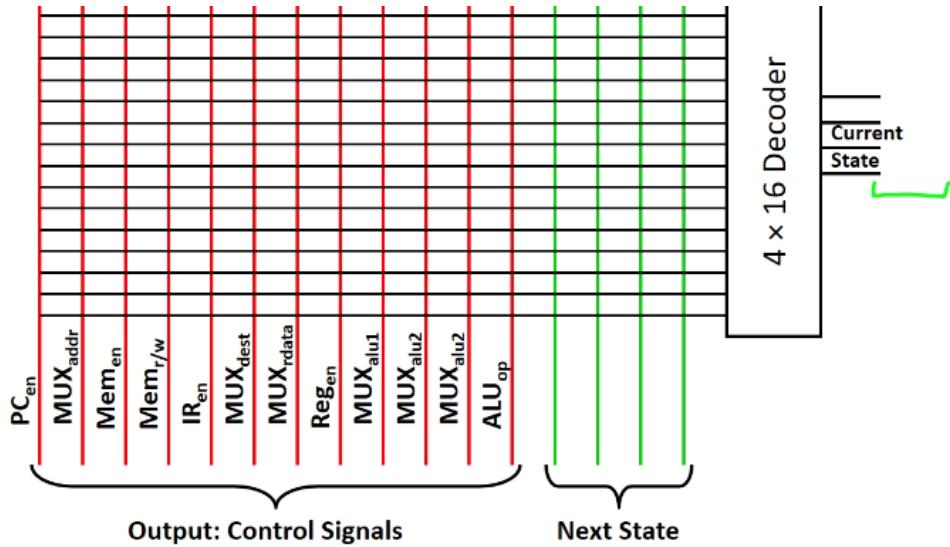


一共 13 个 state，所以需要 4 个 Bits 来 encode state

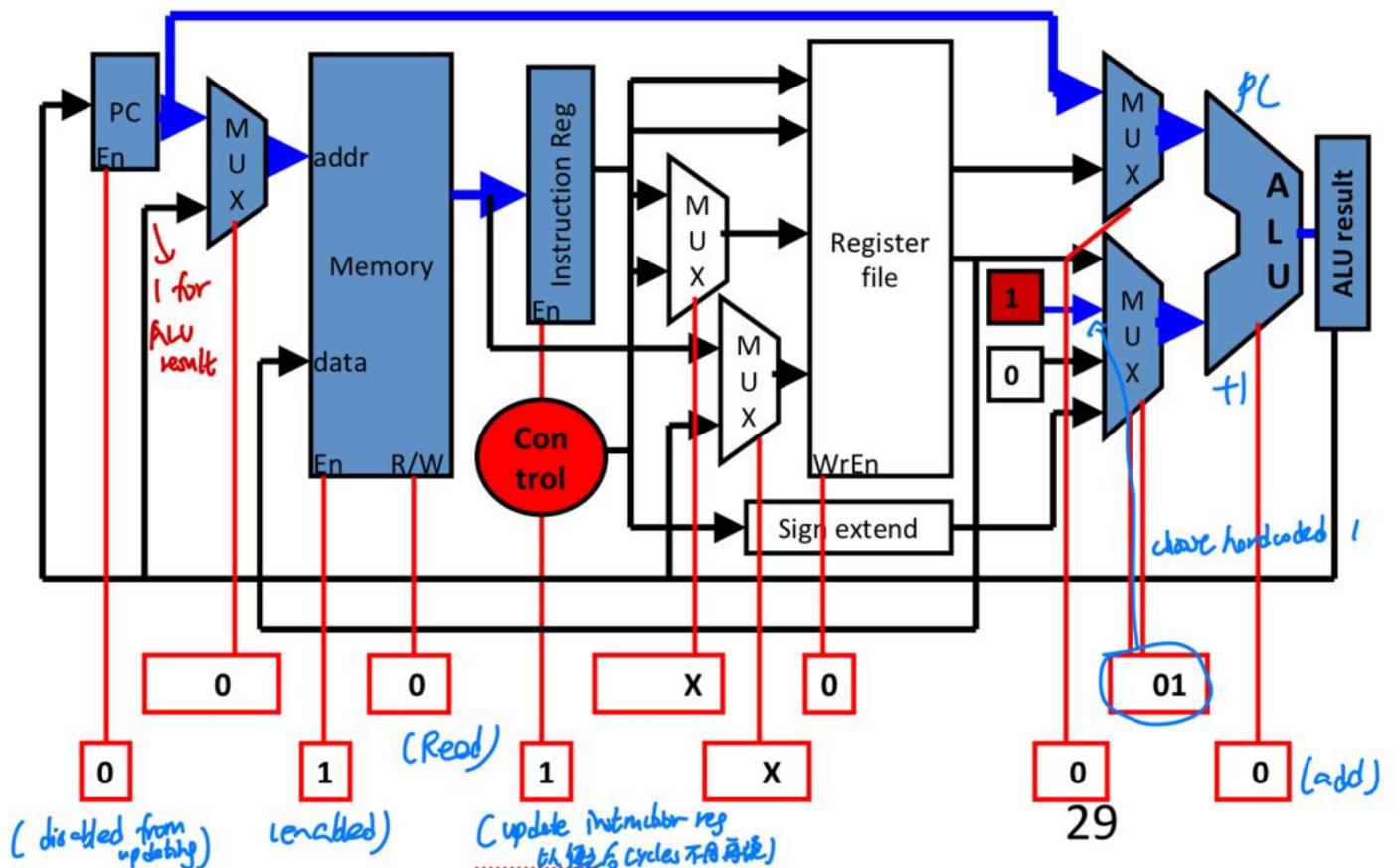
(为什么 lw 要分成三个 states？因为我们希望宁愿多一个 state 也不要 clock cycle 时间变长，不然所有 Instructions 的ns都变长了)

所以 ROM 的大小是 $2^4 \times 12$





State 0 (Universal): fetch



1. 设置 PC_en 位 0, 因为我们这个时候还不想 update PC
2. 设置 MUX_address 为 0, 因为我们想要 PC 的 input 而不是结尾 ALU 返回回来的 Input
3. 设置 memory enabled 为 1, 要读 instructions
4. 设置 Mem_rw = 0, read

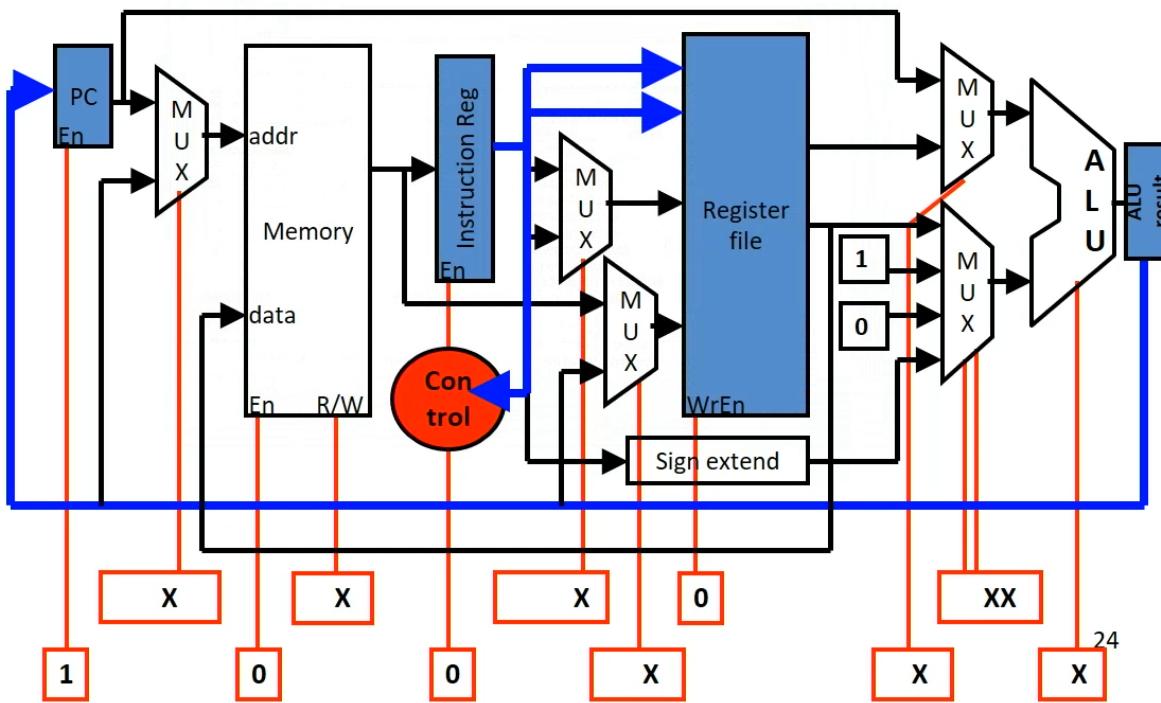
- 设置 instruction reg (改写)enabled = 1, update 一次 instruction reg, 之后的 cycles 就都不用 read instruction 了
- 设置 MUX_alu2 为 01, 这样就切换到了我们 hardcoded 的 1, 把 PC + 1 传输到了 ALU 里

State 1: Decode

decode 阶段做的事情就是：更新刚刚++的PC；read regA, B；通过 opcode determine next state.

只需要设置 PC_en = 1 表示 update, memory enabled = 0 表示这次不读 memory, instruction reg update enabled = 0, reg file Wr_en = 0 就可以，其他都无所谓

**Update PC; read registers (regA and regB);
use opcode to determine next state .**



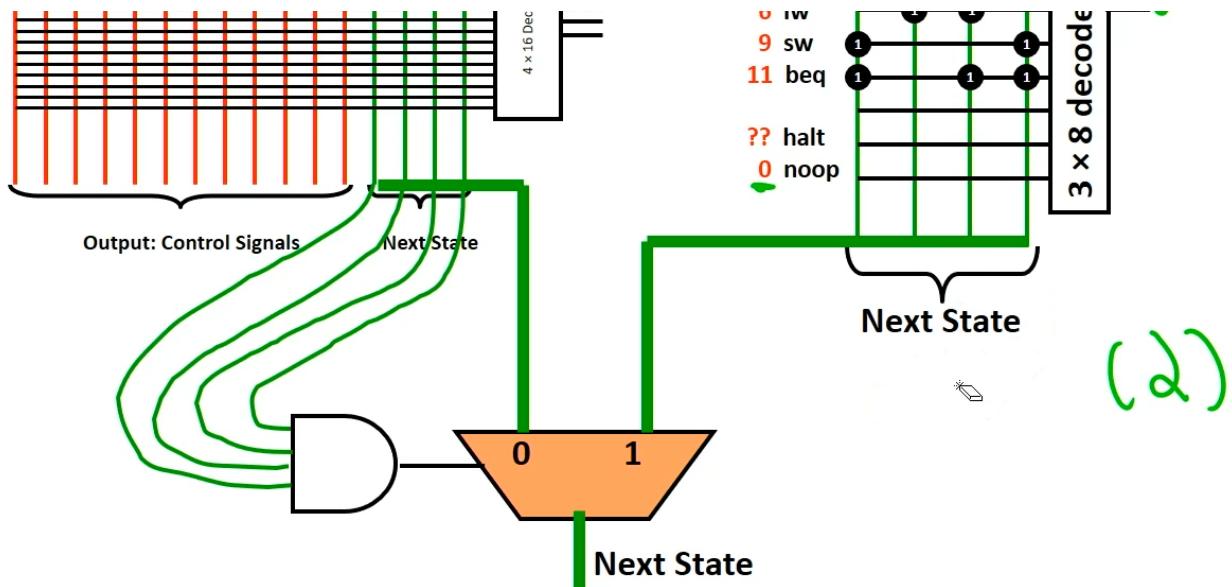
如何确定下面一个 state 是五个里面哪一个？

我们把 3 bit 的 opcode 看作 control rom 的 extra input.

question: 这难道不会让 ROM size 乘以 8 吗？

答：使用 combinatorial logic 避免一下。





39

做法：当且仅当我们在 **fetch state** 的时候，设置原始的 **next state** 为 1111 (值为15，并没有这个 **real state**)，把这四个 1 进行一个 AND gate，作为一个 mux 的 choice bit。

我们通过原始 next state 和一个由 opcode 数据决定的 next state 表进行 MUX 选择。

在其他 state 下，我们的 Next state 都是 0-12 之间的数，所以由 AND 得到的 choice Bit 是 0，进行正常过 state；当且仅当我们在 **fetch state** 时，这四个 bits 是 1，于是右边的 **opcode decoder** 表编辑的 ROM 决定了下一个 state。

State 2-3: Add

State 2: 在 decode 阶段我们从 instruction 中确定了需要读取的两个 reg 是哪两个

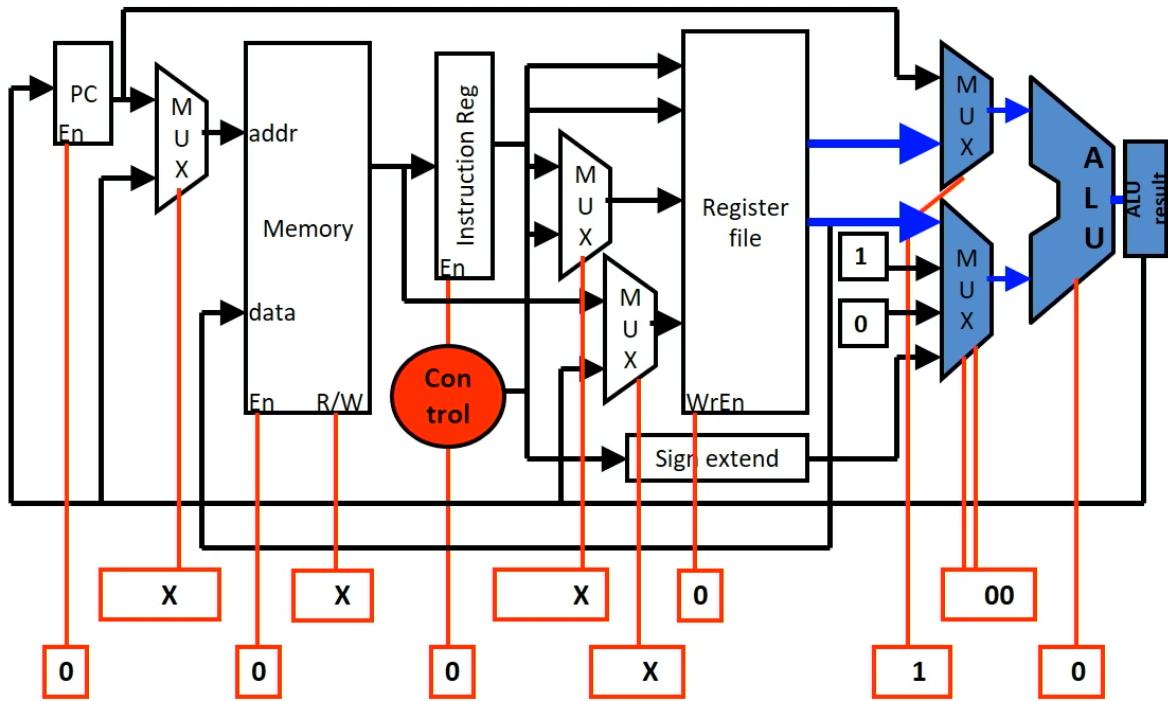
State 2 中我们把 reg file 里这两个 regs 的值输出到 Mux_alu1 和 Mux_alu2 里

所以 Mux_alu1 设置为 1, Mux_alu_2 设置为 00,

ALU 设置为 0, 以取 Add

其余都随意。注意到 PC_en, memory_en, Inst reg_in 应该为 0, 不需要再跑一次

Send control signals to MUX to select values of regA and regB and control signal to ALU to add



next state: 3

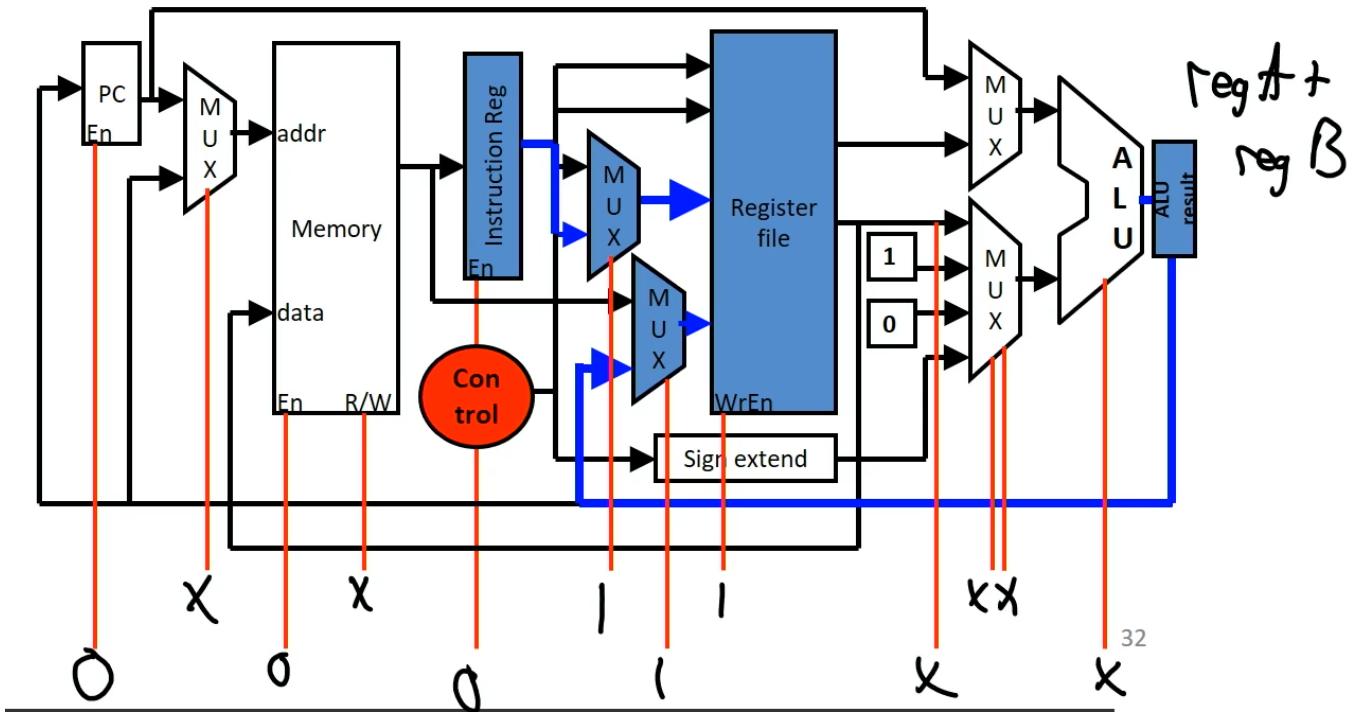
state 3: 现在 regA + regB 的结果已经进入了 ALU result, 最后一个 cycle 要做的事情就是把它传到 reg file 前的 mux,

Note: reg 前面两个 MUX, 从 Inst reg 来的 mux 是选择 destR 的, 上面表示选择 inst 的 18-16 bits, 下面表示选择 2-0 bits, 由于我们的 add 的 destR 在 2-0, 我们选择为1

从 ALU result 来的 MUX 表示接受 memory 的 read 还是接受 ALU result 的 read, 我们接受 ALU result 的 read, 选择 1

于是:

Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.



State 4, 5 nor 和 add 基本差不多。我们可以分辨。

State 6-8: lw

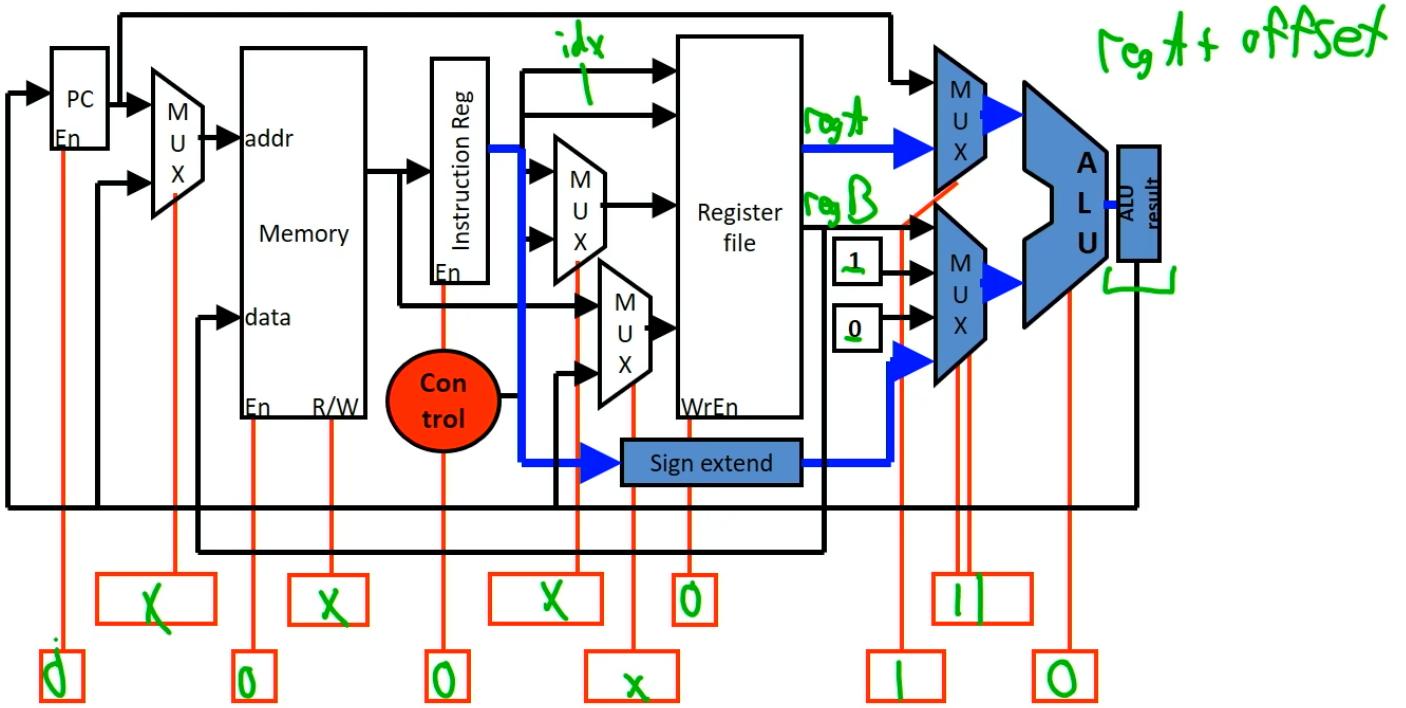
State 6: 计算出 regA + offset

这个时候我们的 MUX_alu1 应当设置 1，因为我们想要 take regA 而不是 PC 的输入

MUX_alu2 应该设置 11，因为我们不想要 regB 或 hardcode 0,1 的输入而是想要 offsetfield 的输入

然后 ALU 设置 0，进行 add

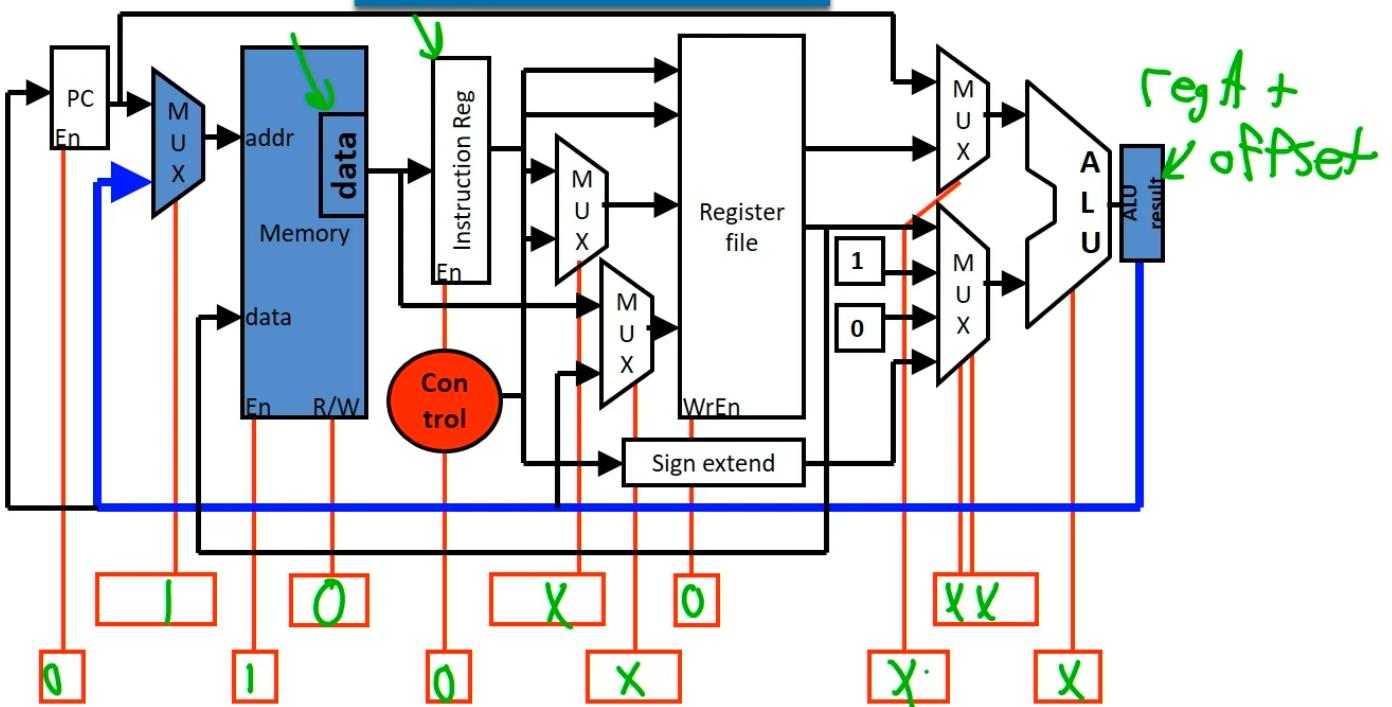
Calculate address for memory reference



State 7: read memory location

Read memory location

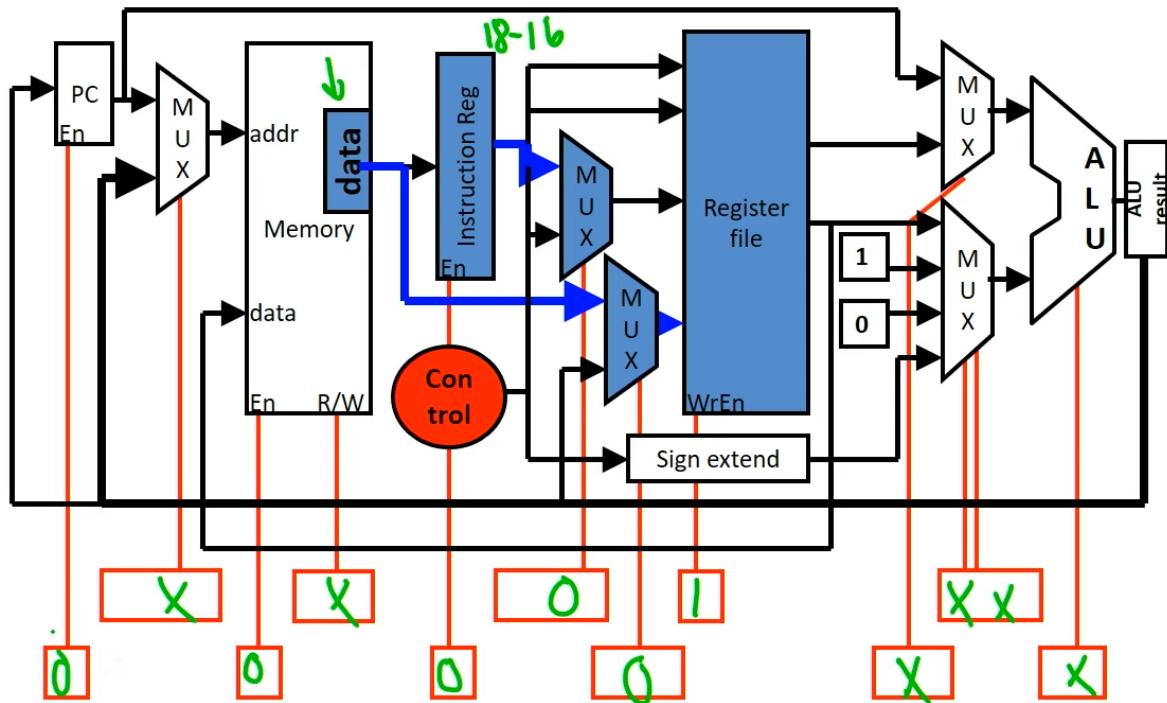
Loaded data stored in "data" reg
(analogous to the Instruction Reg)



Note: 我们这个时候读取了 memory 的某一行地址

State 8: Write memory value to reg files

Write memory value to register file



State 9-10: sw 和 lw 前面基本一样，只是不需要最后一个 cycle 而已

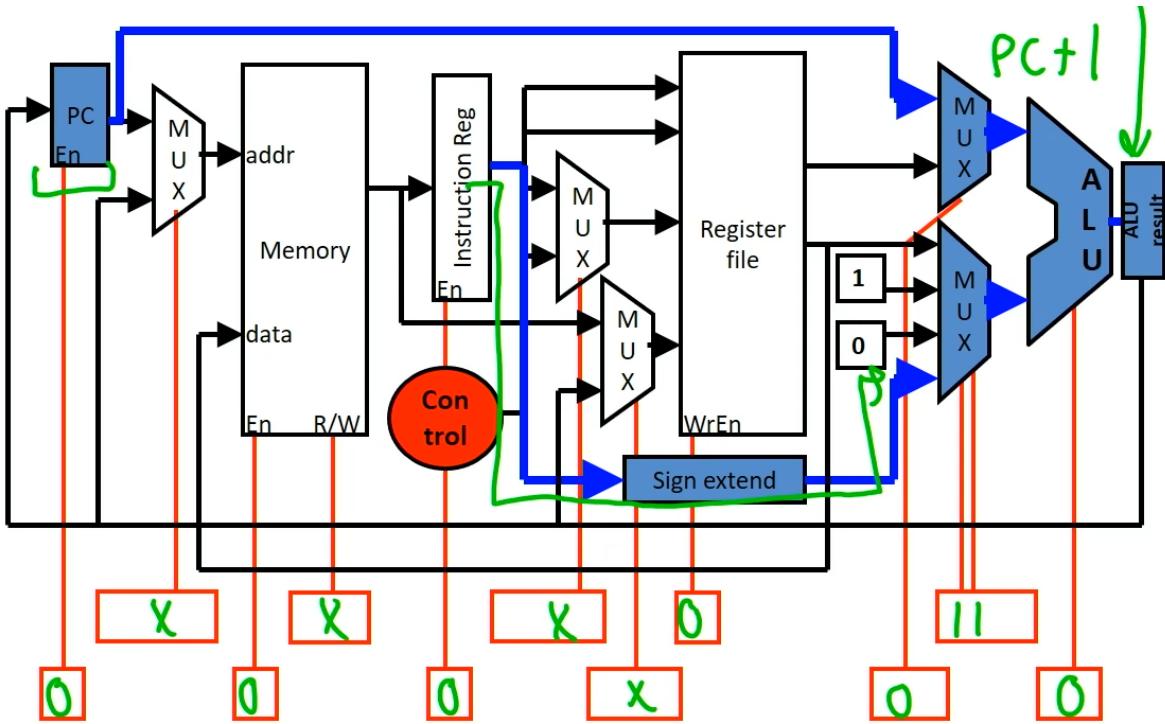
State 11-12: beq

State 11: Calculate PC + 1 + offset 进入 ALU result

和前面一样。我们注意到 PC 此时已经是 PC+1 了，所以直接把 PC 和 offsetfield (MUX_alu2 的 11 选项) 相加就可以

Calculate target address for branch

$PC + 1 + offset$



next state: 1100

State 12: 判断如 $Data[regA] == Data[regB]$, 我们把 ALU result 里存储的新 PC 值存进 PC

和 single cycle 的逻辑一样, 我们需要一些额外的逻辑门来做到这件事

我们需要把 **opcode** 和 **1100** 进行比较, 并且把比较结果和 **PC** 是否 enable 进行一个 **OR gate**. 当 $Data[regA] = Data[regB]$ 在 ALU 中计算出相同时, 我们另起一个 ALU result, 把另外一个 ALU result 也就是 $PC + 1 + offset$ 的地址传给 PC

这样不影响其他 states 时的行为。

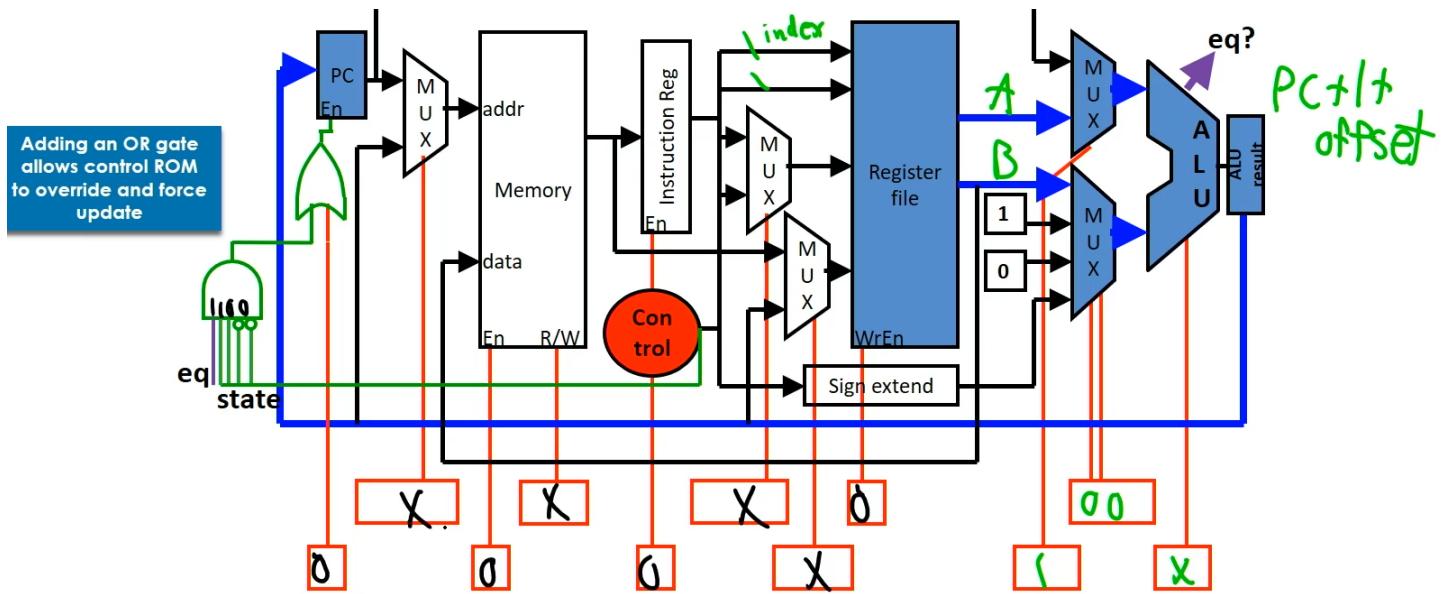
这里有一件比较抽象的事情: 我们此时要改写 PC, 理应设置 **PC_en = 1**

但是我们发现, 我们这个思路就必须设置此时 **PC_en = 0**. 因为如果此时 **PC_en == 1** 那么我们的 OR gate 就导致比较结果没用了, 不论 $Data[regA] == Data[regB]$ 与否都会更新 PC.

所以此时正确的 control bits 是

Beq4:

**Write target address into PC
if ($data_{regA} == data_{regB}$)**



Multi-cycle behavior

我们对于每个循环，仍然是取最大延迟的一个 State 来决定 cycle time

假设我们的延迟表是这样的：

- 1 ns – Register File read/write time
- 2 ns – ALU/adder
- 2 ns – memory access
- 0 ns – MUX, PC access, sign extend, ROM

我们发现：我们设计的 cycles，每一个最多都只有一个 operation. 所以一个 cycle time = 延迟最大的 cycle 延迟 = 延迟最大的 operation 延迟 = 2ns, (read memory)

回忆 single cycle: 一个 cycle 延迟是所有 Instructions 最大的可能延迟 = lw 的延迟 = $2 + 1 + 2 + 2 + 1 = 8 \text{ ns}$

而 multicycle: lw 有 5 个 cycles, take 100ns 延迟；其他指令都take 80ns 延迟

我们发现延迟居然还变大了，真的布什人

但是我们仔细想一想：如果最大延迟的 instruction 不是 lw 而是一个 take 16ns 的 Instruction, 那差距就大了
或者如果我们优化某个 operation 的延迟，那么 single cycle 得到的好处就比 multicycle 小很多了

我们关心的真正问题就是 execution time of a program.

Execution time = CPI * #insts * clockPeriod

CPI 即 average number of clock cycles per instruction

Single Cycle Processor 的 CPI: 1 (但是 clock period 长, 比如 10ns)

Multi Cycle Processor 的 CPI: 4.25 左右 (但是 clock period 短, 比如 2ns)

我们更希望的是 CPI 和 clock period 都短

(next time: pipeline processors

Lec 13 Pipelining

Multicycle 强于 single cycle 仅在有某些指令花的时间相较于其他指令远更长的情况下。如果每个指令的执行时间差不多, multicycle 甚至不如 single cycle.

我们想要一个性能更好的 datapath: multicycle 是一个好主意, 我们可以在 multicycle 的基础上增加一些硬件, 让 datapath 一次可以执行多个并行的 cycle: 当上一个指令运行 cycle 2 的时候, 我们同时运行下一个指令的 cycle 1. 从而, 我们可以通过并行地运行多个 cycles 来达到在单个 cycle 效率和 multicycle datapath 相近的同时, CPI 也接近 1. 这极大提升了性能

这就是 pipeling 的理念.

Implementation Idea

具体 implementation:

1. 我们把 instruction 的运行分成几个 cycles. 和 multicycle datapath 一样.
2. 对于每个 cycle, 我们都设计它自己的 datapath, 将其称作一个 stage.

在 timeline 上的给定时间点, 每个 stage (共五个) 都对应着一个指令的执行.

也就是说: stage 1 在运行第 $n+4$ 个指令的第 1 阶段; ...; stage 5 在运行第 n 个指令的第 5 阶段.

我们创建 **pipeline registers** (一系列 flip flops) 在 stages 之间进行交流. 这样也可以防止 stages 之间相互干扰 (否则如果电流从 stage 1 一直运行到 stage 5 不断进行改写, clock 将难以控制这个精确的时间)

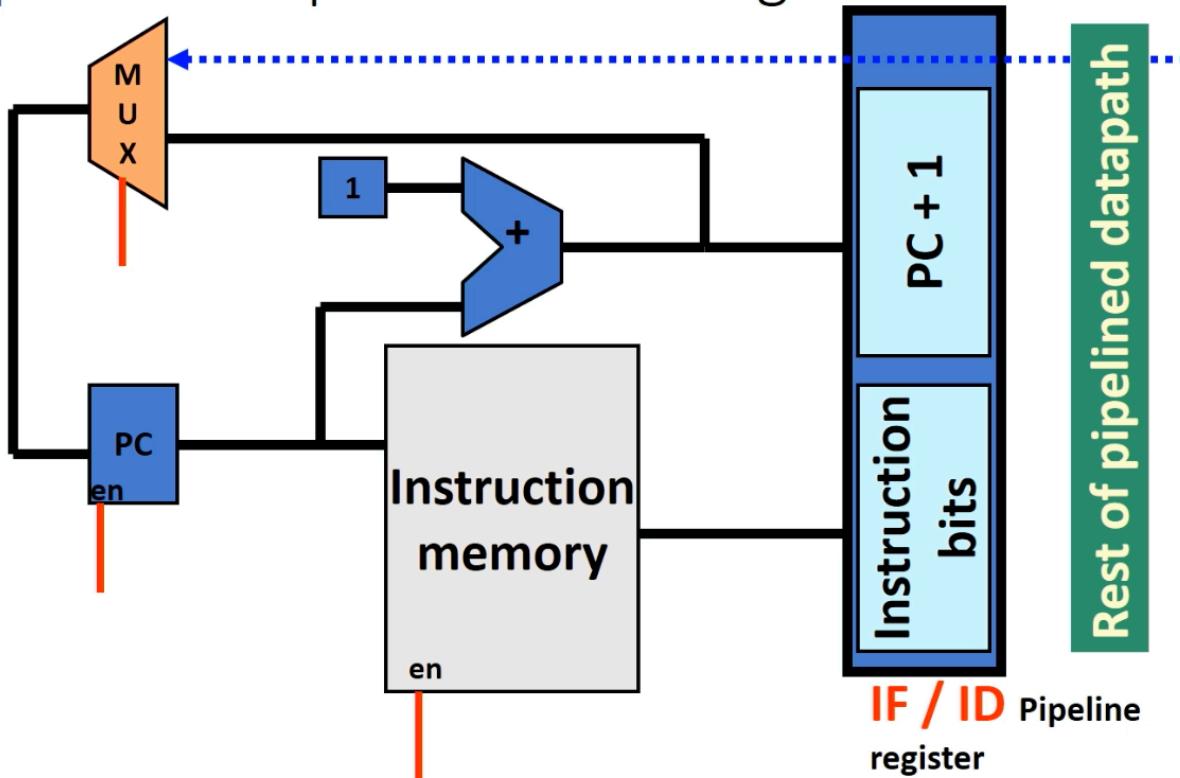
pipeline registers 就像 multicycle 的 instruction register. 只不过在 pipelining 只, 我们对每个 stage 配备一个 pipeline register. (一共有五个)

3. 每过一个 clock, 左边 stage 的指令就带着它更新的信息传递到右边.

Stage 1: Fetch, IF/ID reg

要做的事情：

1. index memory by the address in PC (read inst)
2. PC++ (暂时假设无 branch)
3. 把这两个信息写进 IF/ID reg



这里的 IF/ID reg 表示在 stage1: fetch 和 stage2: decode 之间的沟通 reg.

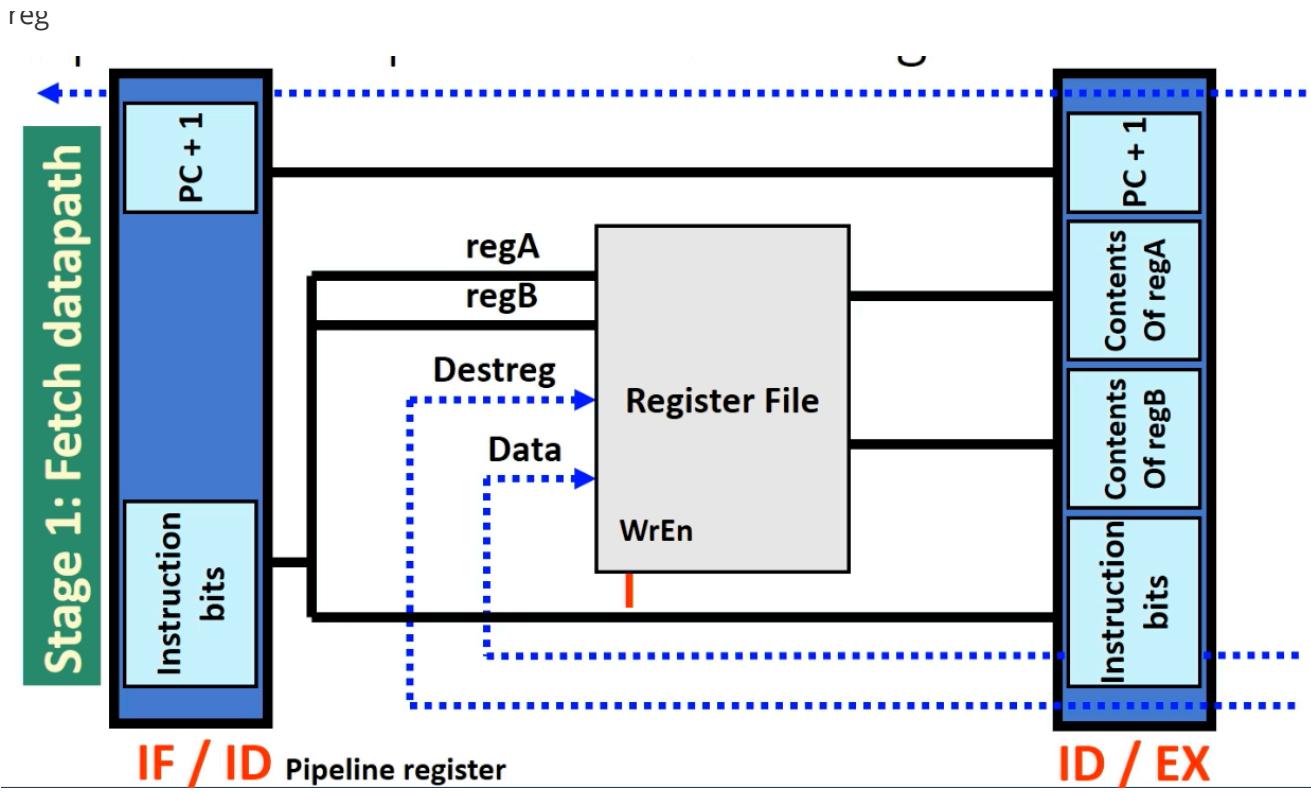
IF/ID reg 应当包含的信息是 instruction bits 32 位, 以及 PC+1 的 32 位.

蓝线表示 Mux 的另一个 input, 来自 later stages.

Stage 2: Decode, ID/EX reg

要做的事情：

1. decode instruction
2. read from reg file (specified by regA, regB of the instruction bits)
3. 把 regA, regB 的信息, 连带 IF/ID 里面 PC 以及 instruction bits (只需要 0-2/16-18 以及 opcode) 一起传到 ID/EX



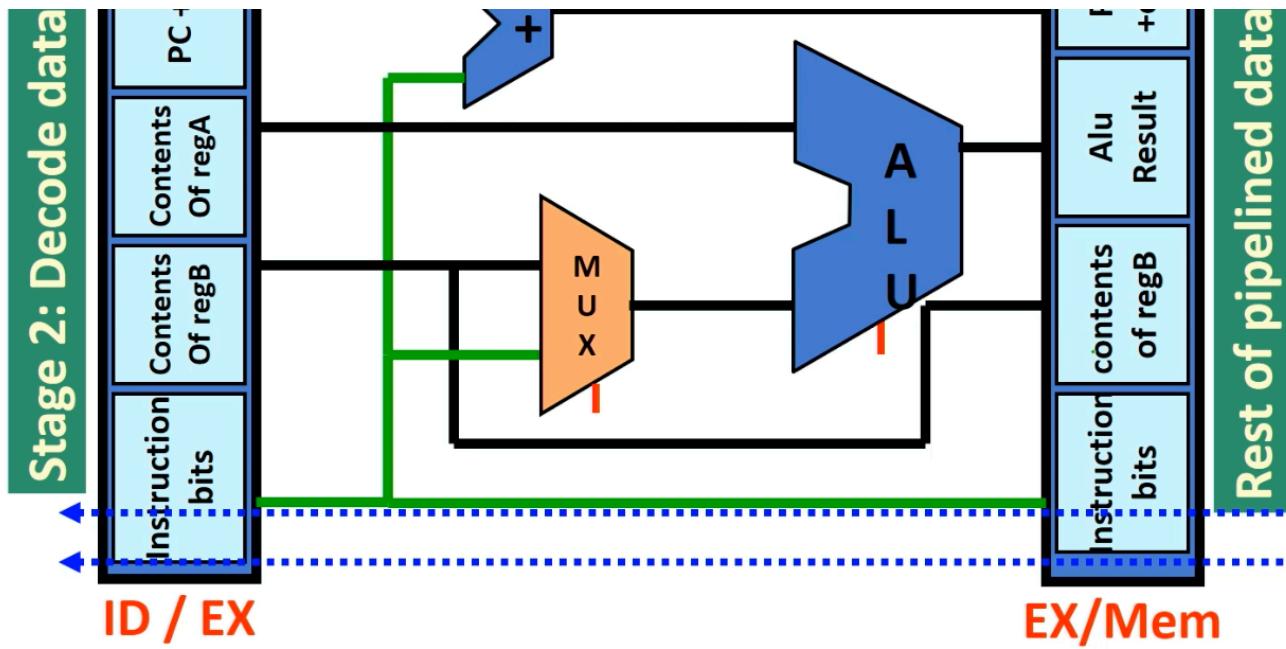
Stage 3: Execute, Ex/Mem reg

execute 也就是运算. 我们的运算只有这几个情况: add, nor; lw/sw 算地址; beq 算 equal; pc+offset

要做的事情:

1. 运算要么是对 regA content 和 regB content 要么是对 regA content 和 offset 进行. 我们把 regB content 和 inst bits 中的 offset 加一个 mux, 与 regA content 过一个 ALU.
2. PC 和 inst bits 中的 offset 过一个 +
3. PC 结果, regB content 和 ALU 结果以及 inst bits 传给 stage 4.





Stage 4: Memory Op, Mem/WB reg

这个 stage 是专门给 lw/sw/beq 的. 其他 opcode 都会通过 enable bit 忽视这个 stage 的 memory read data 并且把 inst bits 和 ALU result 原封不动传递下去 (比如 add/nor)

要做的事情:

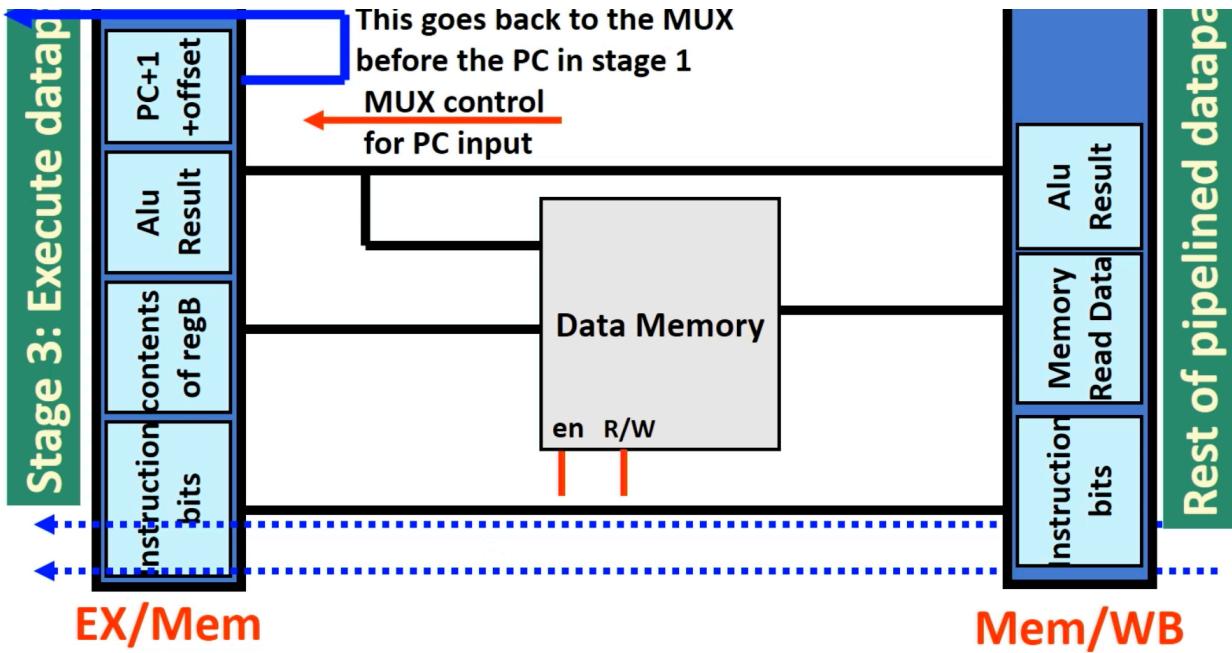
1. 把上一步加上 offset (if not 0) 的 PC value 送回 stage 1 上的 PC reg 中.
2. 对于 ALU result (regA+regB), 我们始终保持这个 result 但是其实只有 add/nor 指令下这个 result 有用: regC 的值
3. 把 ALU result 和 regB 的 content 带进 data memory:

如果是 lw, 那么 enable read bit 去读 data memory, 把 **data memory** 里查到的 ALU 地址对应值放进 **Mem/WR stage reg** 里作为 **memory read data**, 以待下一步 write back to reg

如果是 add/nor, memory read data 可以忽略.

如果是 sw, 那么 enable write bit 去写 data memory, 把 regB 的 content 写到 data memory 中 ALU result 的位置.

这次不用再存 PC value 了, Mem/WB 是最后一个 stage reg.

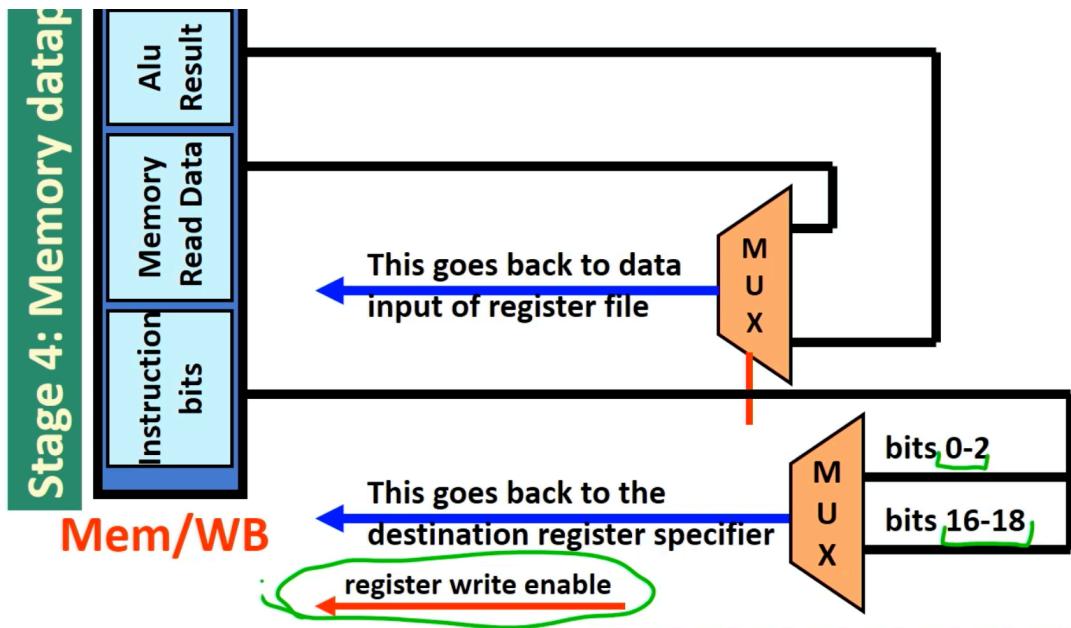


Stage 5: Write Back

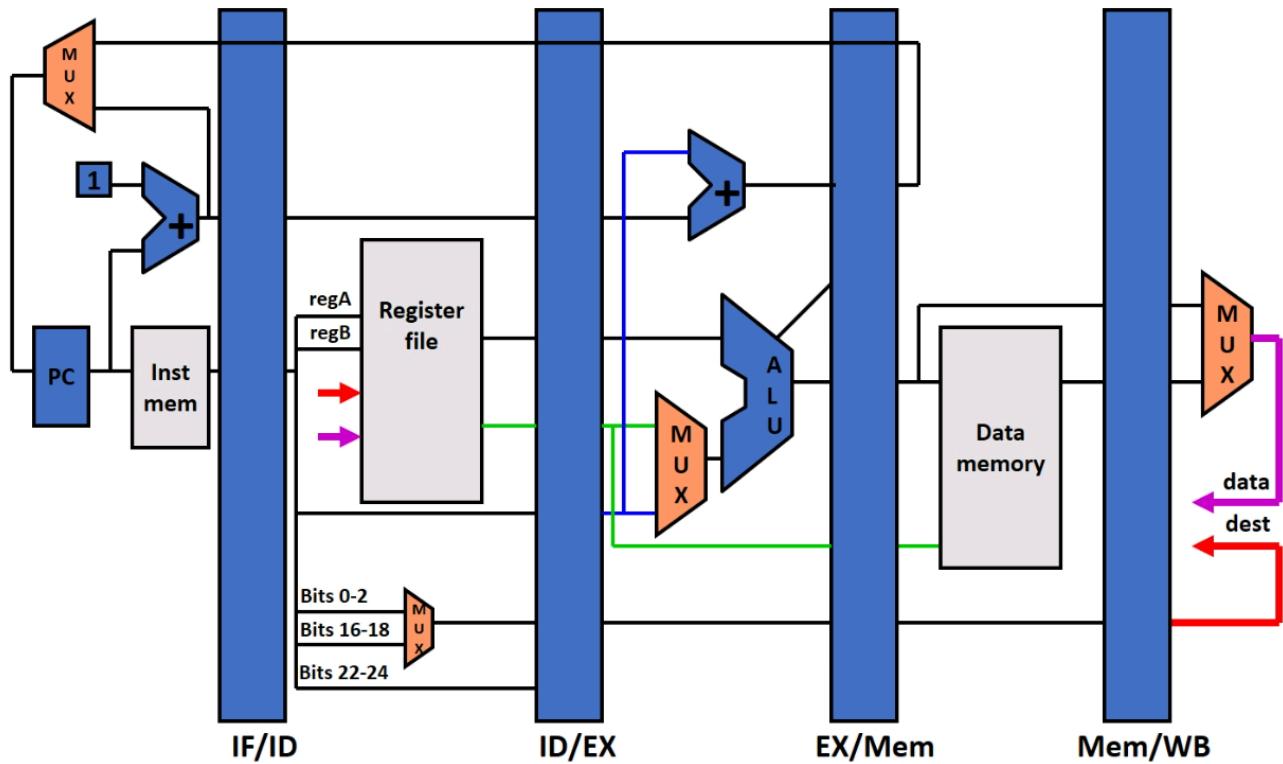
要做的事情：

1. For lw, 我们要把 "memory read data" 写回 inst bits 指定的 regB 去. (上面的 Mux 传值, 下面的 mux 传进入哪个 reg, 即 bits 16-18)
2. For add/nor, 我们要把 ALU 结果写回 reg C (bits 0-2) 去. (上面的 mux 传值, 下面的 mux 传进入哪个 reg, 即 bits 0-2)

Pipeline datapath – Writeback stage



Overview



Lec 14 Data Hazard

我的评价是在接受同时运行多个指令这个理念的时候我们首先就会想到两件事：

1. read reg 发生在 stage 2, write back 发生在 stage 5.

如果我 stage 3 的 inst 是改变 reg 的，那么它后面 stage 2 的 reg 读到的理应是 stage 3 的 inst 未来在 stage 5 修改过后的 reg value. 但是它却读到了修改前的. 这是一个数据冲突啊

(这种隐患叫做 **data hazard**)

2. 一个 branch instruction (beq/jalr in our LC2K) 会改变 PC 的值，但是 PC 的值只有在 stage 4 才会改变.

如果我们的 beq 在 stage 3, stage 2 fetch 的理应是 PC 更改后的 instruction. 但实际上按照我们现在的 implementation 我们 fetch 到的是 PC 更改之前的 instruction. 这是一个指令错读

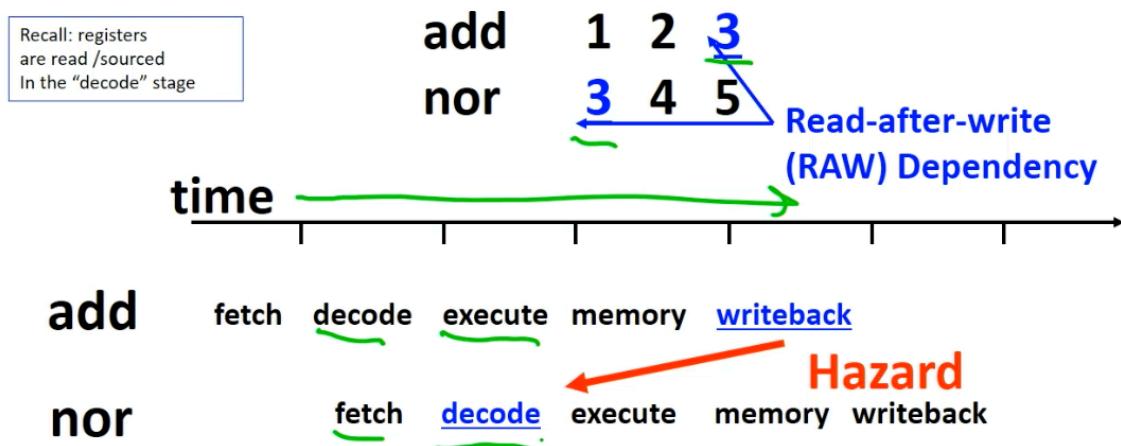
(这种隐患叫做 **control hazard**)

这节 lec 我们讨论 data hazard.

Recall: 我们的 pipelining 有五个 stage: fetch, decode, execute, memory op, writeback.

Data Hazard 和 Data Dependency

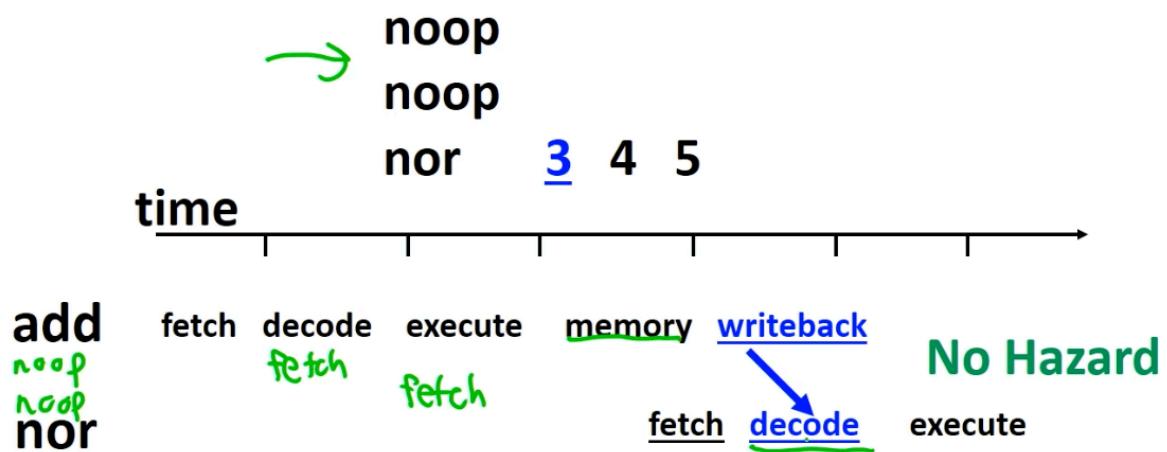
以下为一个 **Data Hazard**: 出现 decode 的时候，该 decode 的 reg 有理应完成但当前 time 并未完成的 write back.



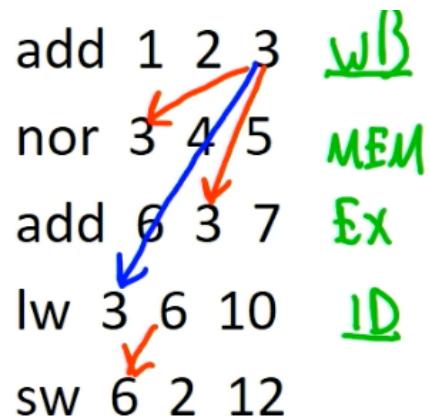
If not careful, nor will read a stale value of register 3

以下作为对比为一个正常的 data dependency

add 1 2 3



Ex2: 蓝色表示 data dependency, 红色表示 data hazard



Method 1: 加入 noop 来避免 data hazard

只要确保我们的 instruction 里面没有 data hazard 就好了.

天才!

实际操作是我们在 dependent instructions 之间插入 noops. 因为 decode 和 writeback 差了三个 stages, 所以我们插入至多两个 noops 就可以了.

也就是说我们指望 compiler 和 assembler 具有 detect data hazard 的能力.

add 1 2 3 — write register 3 in cycle 5
noop



好处是我们不需要更改任何 hardware.

但是问题是：

当新的 processor 出现时，我们总是希望在上面能跑我们旧的代码. 所以

(1) 所有代码都要重新 compile, 重现插入 noop. 如果代码数量很多, 那么耗时太大

(2) 有时候我们甚至没有 source code 只有 .exe 文件. 我们无法 rewrite it.

并且, 这还会使得 program 变得更大, 25%-40% 的 instructions 都变成了 noops

并且 program execution 更慢了.

Method 2: Detect and Stall

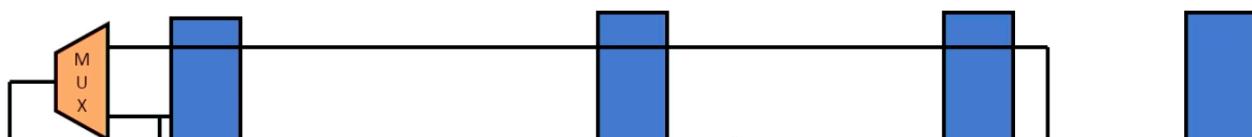
2. Detect and stall: 检测到 hazard 时, stall the processor until the hazard goes away

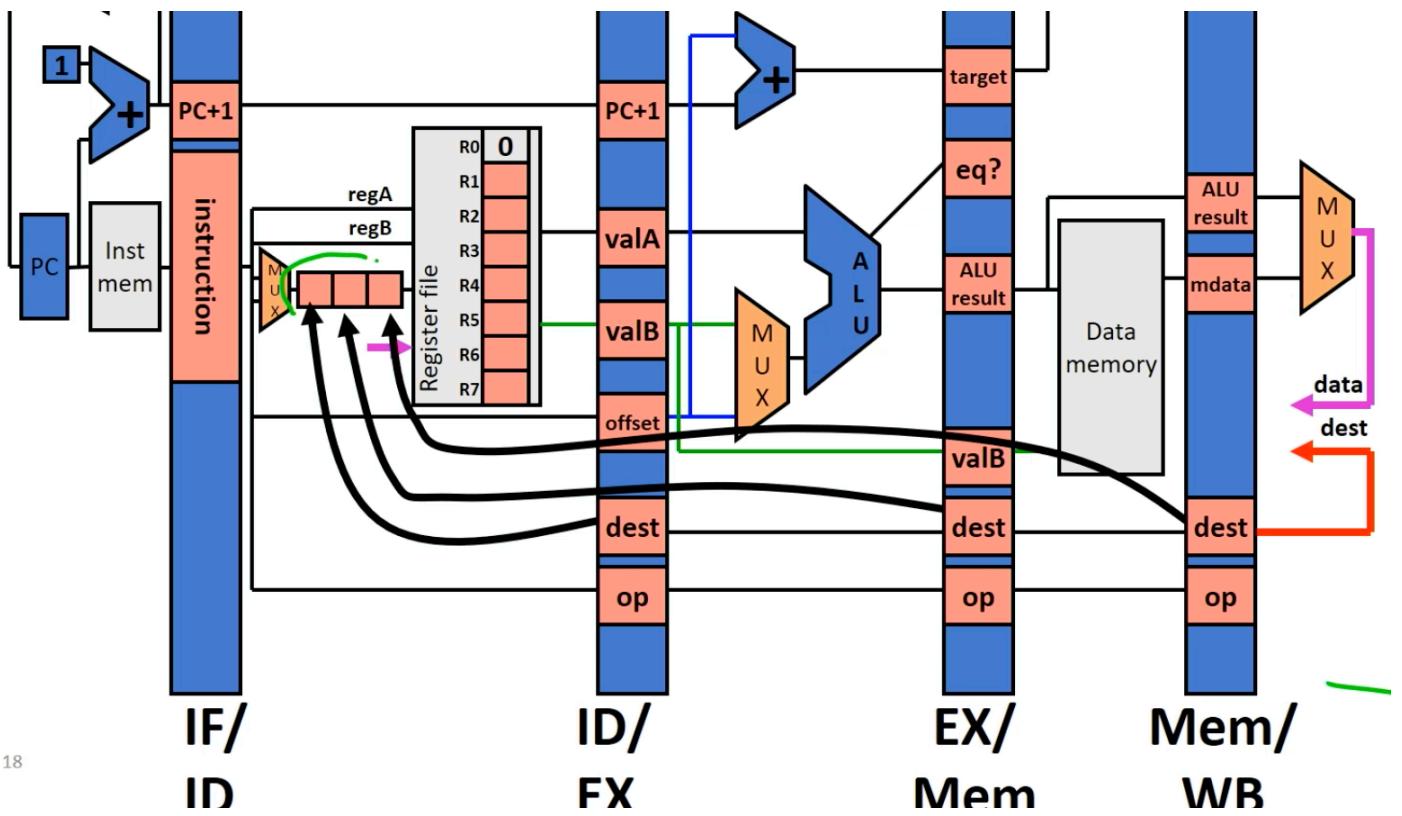
其实和第一个基本是一样的原理。只是第一个是主动修改汇编代码，而 detect and stall 中我们通过调整硬件来实现这个功能

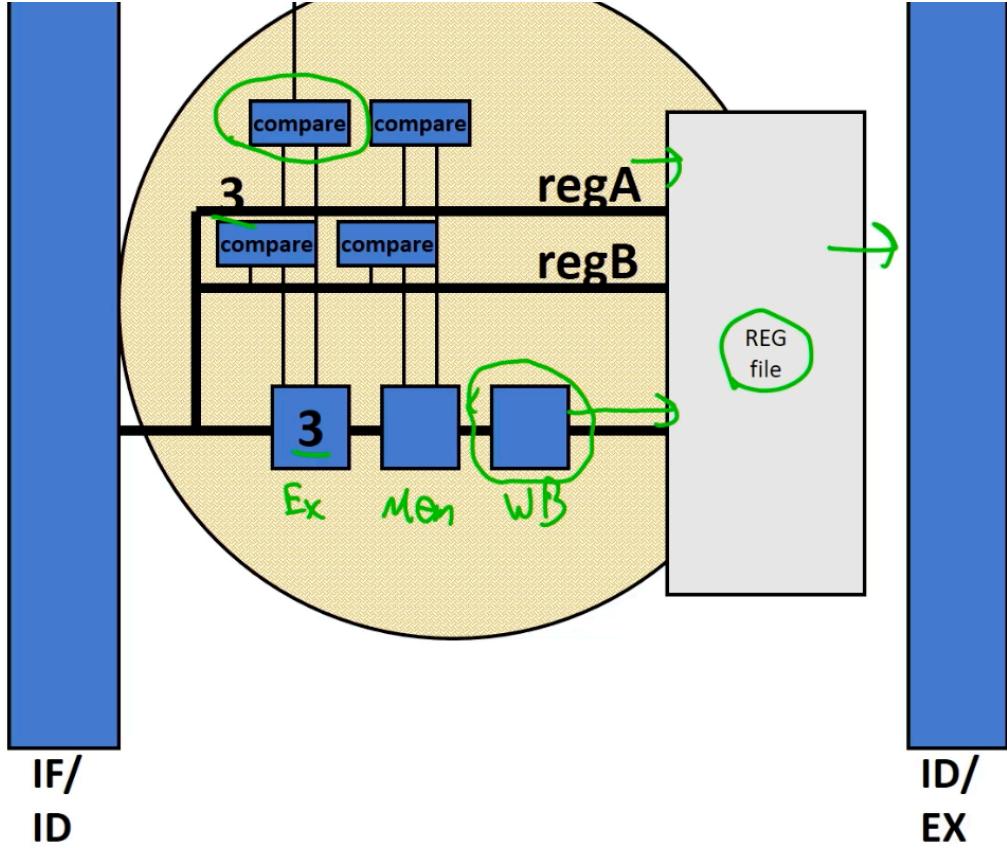
Detect: compare regA, regB with previous destRegs.

Stall: 把 current instructions 留在 decode stage 不往前走

把一个 noop pass 到 execute stage.







实现方法：

让我们在 Decode stage, write back 前加上三个 3 bits 的 latch 块. 我们每个 clock, 都把 ex, mem, wb 三个 reg 上的 instruction bits 中的 "dest reg" 传给这三个 latch 块.

当检测到前两个 ex, mem latch 块上的 reg num 和此时 instruction 的 regA, regB num 相同时, 那么就 detect 了一个 hazard.

(注意, 如果是 wb latch 块上的 reg num 和此时 regA, regB 的 num 相同, 那么不算 data hazard! 因为我们认为, write back 的速度比 regA, regB 的寻址速度更快, 这个时候我们在寻址前已经 write back 了, 所以并不是一个 data hazard.)

当 compare 检测到 hazard 时就停止继续读指令, 而是停滞住此时正在 fetch 以及 decode 的指令, 一直 impose noop 直到不再检测到 hazard 为止.

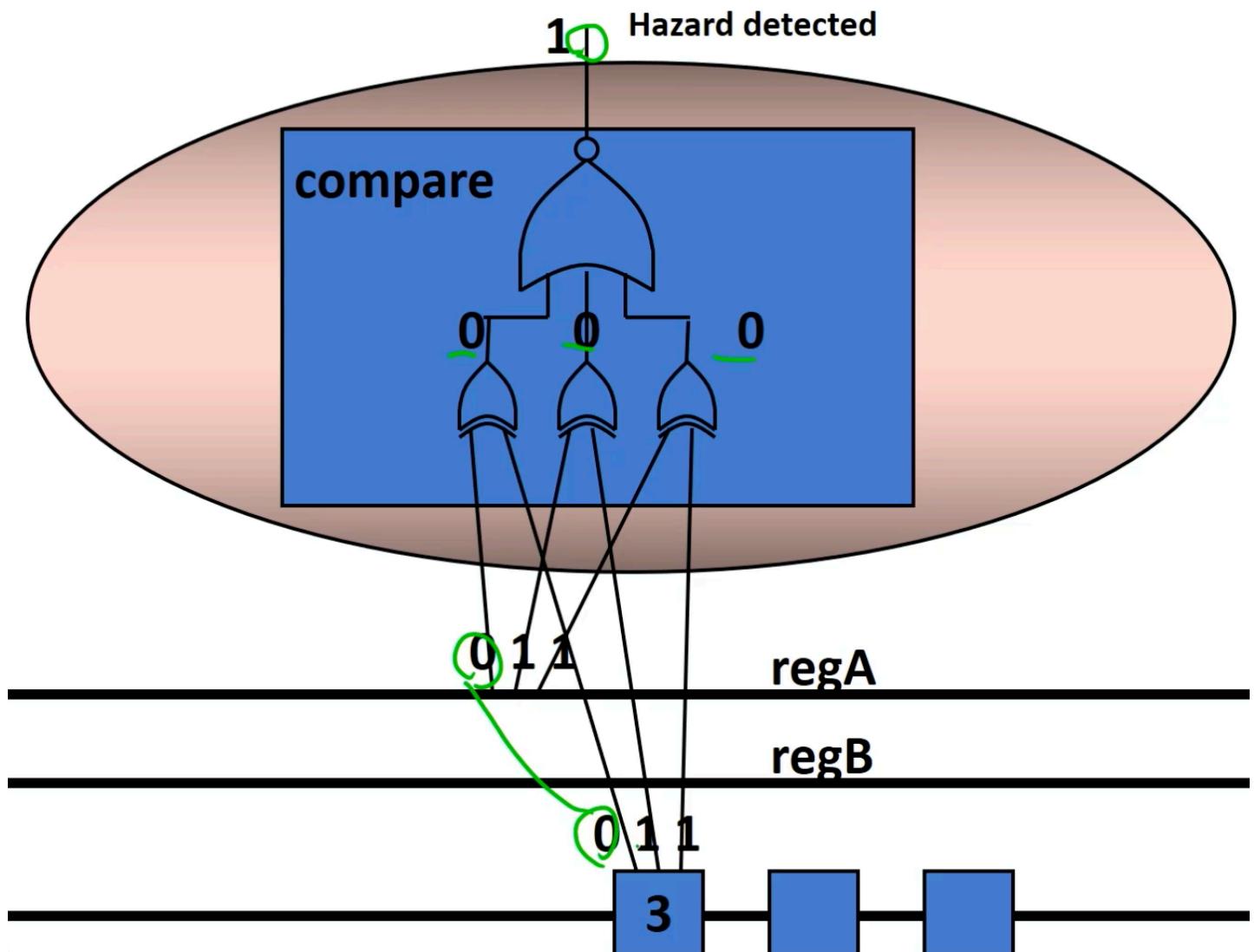
只要 dependent inst 打了 write back stage, hazard 就算解决了。所以, 对于 dependent 在前一个指令的 data hazard, 我们只需要插入两次 noop, 前一个指令就到 Stage 5 Writeback 了; 对于 dependent 在前两个指令的 data hazard, 我们只需要插入一次 noop.

compare 具体实现

compare: 通过对两个 3 bits reg 的每一位进行一个 xor, 表示其是否不同.

最后再把三个 xor 结果 nor 一下.

nor 结果为 1 当且仅当三个 xor 都是 0 (都相同). 这个时候 hazard detected = 1.



add 1 2 3	IF	ID	EX	IVIE	WB								
nor 3 4 5		IF	ID*	ID*	ID	EX	ME	WB					
add 6 3 7					IF	ID	EX	ME	WB				
lw 3 6 10						IF	ID	EX	ME	WB			
sw 6 2 12							IF	ID*	ID*	ID	EX	ME	WB

这个方法比起方法 1，好处在于

1. 修复了 backwards compatibility
2. 不会使得 program get larger

但是 program get slower 的问题仍然存在.

(总体而言，完全强于方法 1.)

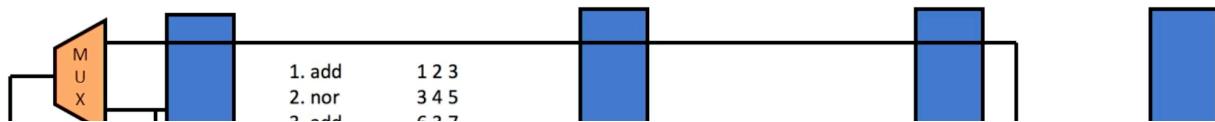
Method 3: Detect and forward

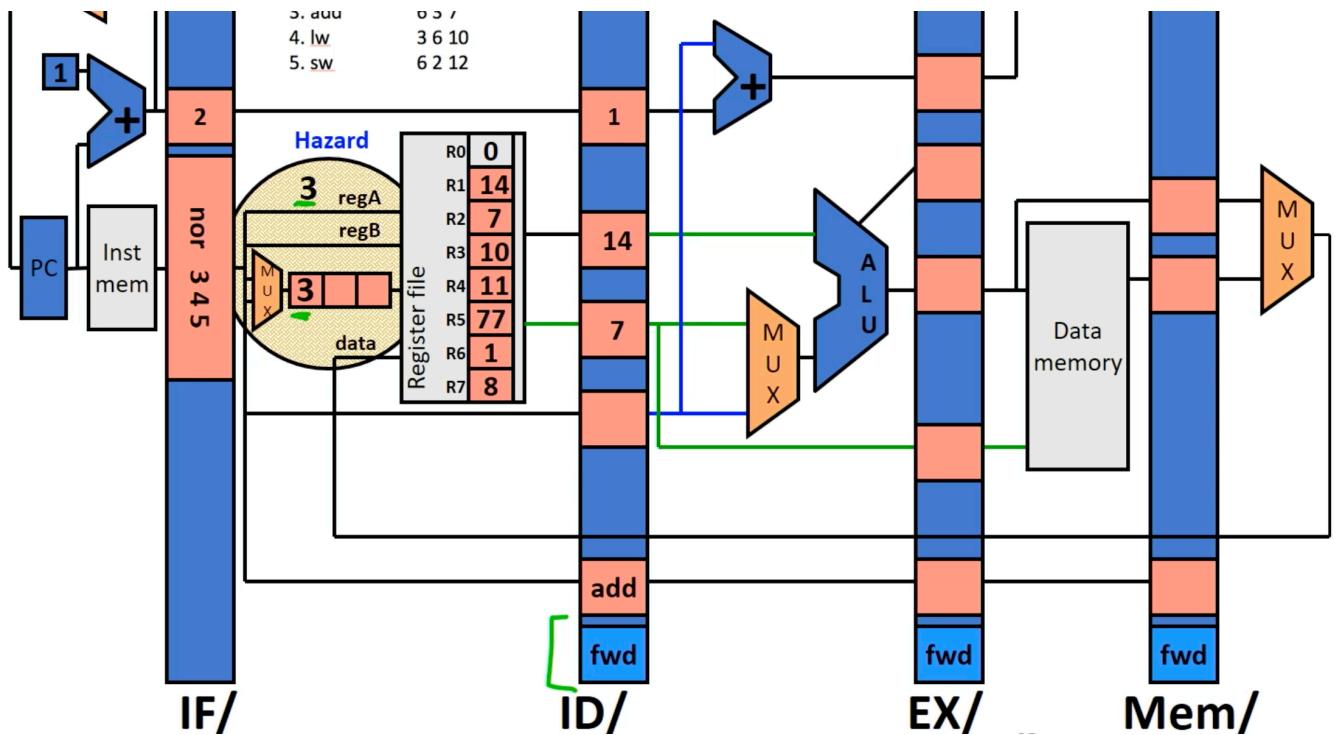
3. Detect and forward: 检测到 hazard 时，fix up the pipeline to get the correct value (if possible)

detect and forward 的 detect 和 detect and stall 里的 detect 不同的点在于四种 data hazard 必须被 treated differently.

Forward: New bypass datapaths route, 把 data 导到它被需要的地方. 我们需要新的 MUX 和 control 来找到这些 data.

做法：在每个 stage reg 上放一个 extra register





这个 extra reg 储存:

- (1) 现在是否有 hazard
- (2) 一个 hazard number: 每一个表示一种 hazard 的 sol —— where to grab it from, where to send it to.

Data Hazard 的所有类型

一共有四种 Data Dependency:

1. RAW: Read after write, 又称为 flow dependency, 即 write 的数据还没有被导入应到的 reg 就被读
2. WAR: Write after read, 又称为 anti dependency。它可能会导致 hazard: 这是一件很逆天的事情。write 的时间肯定比 read 要晚, 既然 read 的指令在前面, 怎么可能发生冲突呢, , 根据资料来看传统的 Pipeline processor 是肯定不会出现这种情况的所以可以放心。因为我们的 LC2K 不仅传统而且简易。现代的 multi-thread processor 会议及乱序处理的 processor 会出现这种问题.
3. WAW: write after write, 又称为 output dependency. 他也可能导致 hazard. 同样不会出现在传统 Processor
4. RAR: read after read. 在任何情况下都不会出现 hazard.

所以我们的 LC2K 的 data hazard 只涉及: read after write.

Note: hazard 至多差距两个 stages。要么是 regA 是另一个的 dest reg, 要么是 regB 是另一个的 dest reg.

举例: 比如这里, nor 的 regA 是 ahead 它 1 stage 的 dest reg. 理应 21, 当前 10. 我们用 H1 来 encode 这种 hazard.

但是, lw/sw 确实有一种另类的 data hazard:

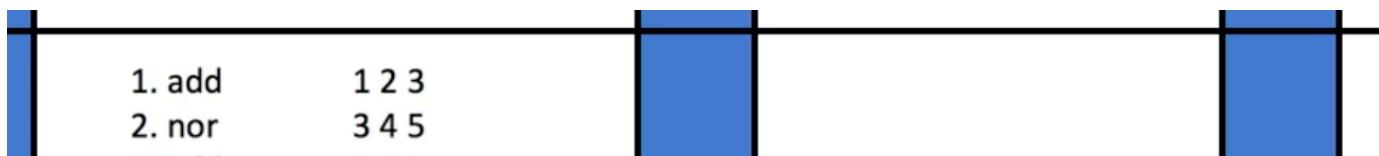
1. **sw 的 regA 是前面的 lw 存储进去的 reg.** 比如这里的 4,5. sw 在 Stage 2 就应该读数据了, lw 的结果在 Stage 5 才会 write back. 这里有三个 Stages 的差.
2. lw 之后紧跟使用被 load 的 reg. lw 在 Stage 4 才真的 load, load 行为会改变 reg 的值; 而后面的 inst 比如 add 如果正好使用应该 Load word 进入的 reg1, 那么它在 Stage 2 读取到的是 load word 之前的 reg1.

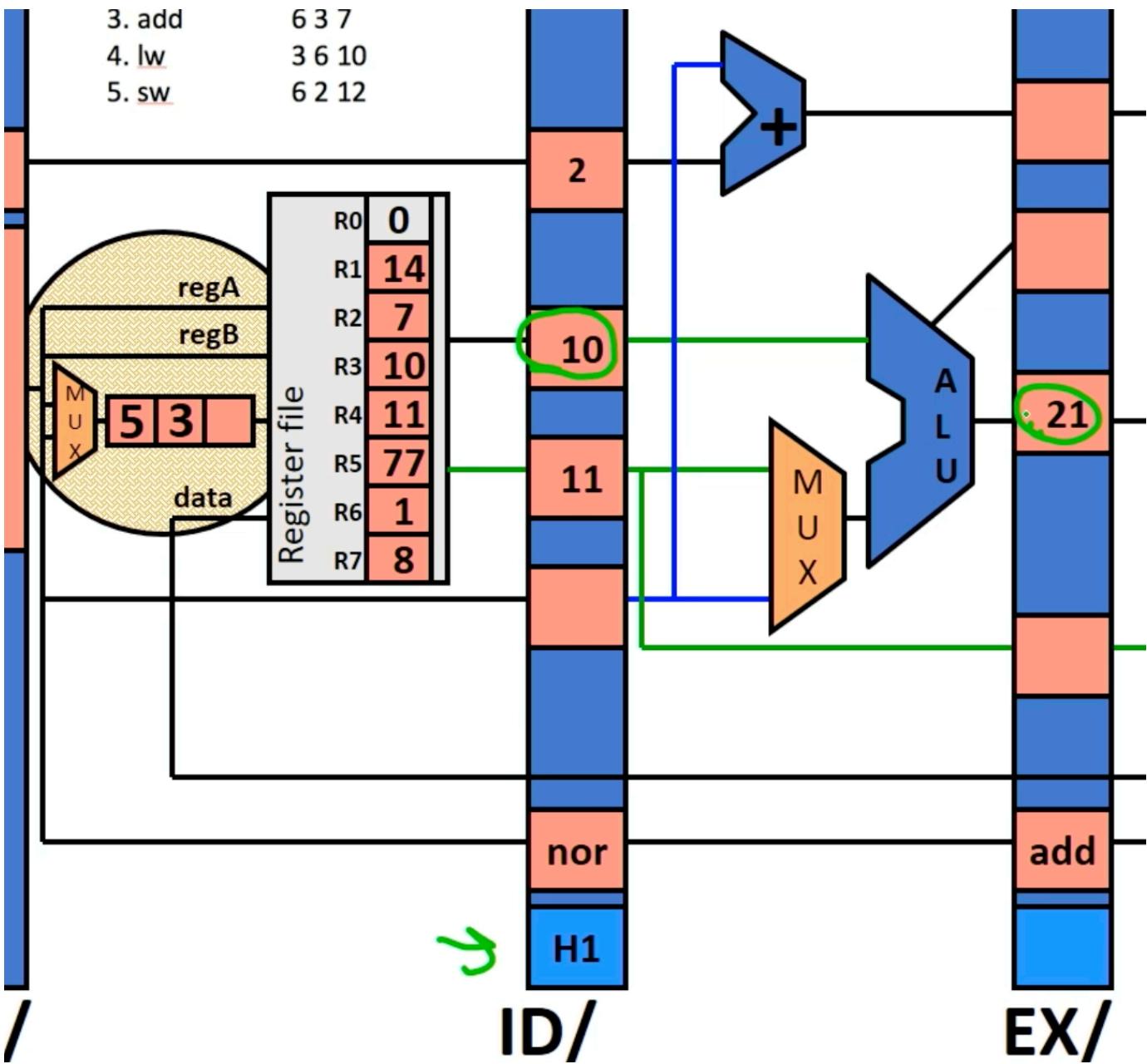
然而代码的意思是, next inst 使用的理应是 load 之后的 reg.

sw/lw 这两种 hazard 的解决方案都是统一: 差距两个 Stage 则直接从 Stage 5 导数据进入 Stage 3; 差距一个 Stage, 只好不得不 stall for one cycle. 见: 处理 sw/lw 的 data hazard.

(plus: 我考虑过的另一个问题是: 如果前一个是 store word, 后一个 load word 的地址正好是刚刚 store 的地址呢? 这会不会造成一个 hazard。。但是仔细想想就多虑了, 因为 memory operation, 读数据是 Stage 4 的。所以不论如何, 等到 lw 真的读数据的时候, 要么 sw 早就已经存了, 要么 sw 正好是 Stage 5, 但是 write back 的速度更快, 刚好比 lw 读取早存好.)

所以一共只有这四种 **data hazard** 类型.



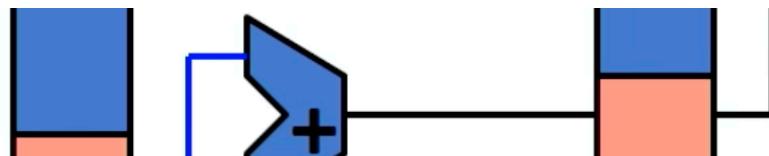


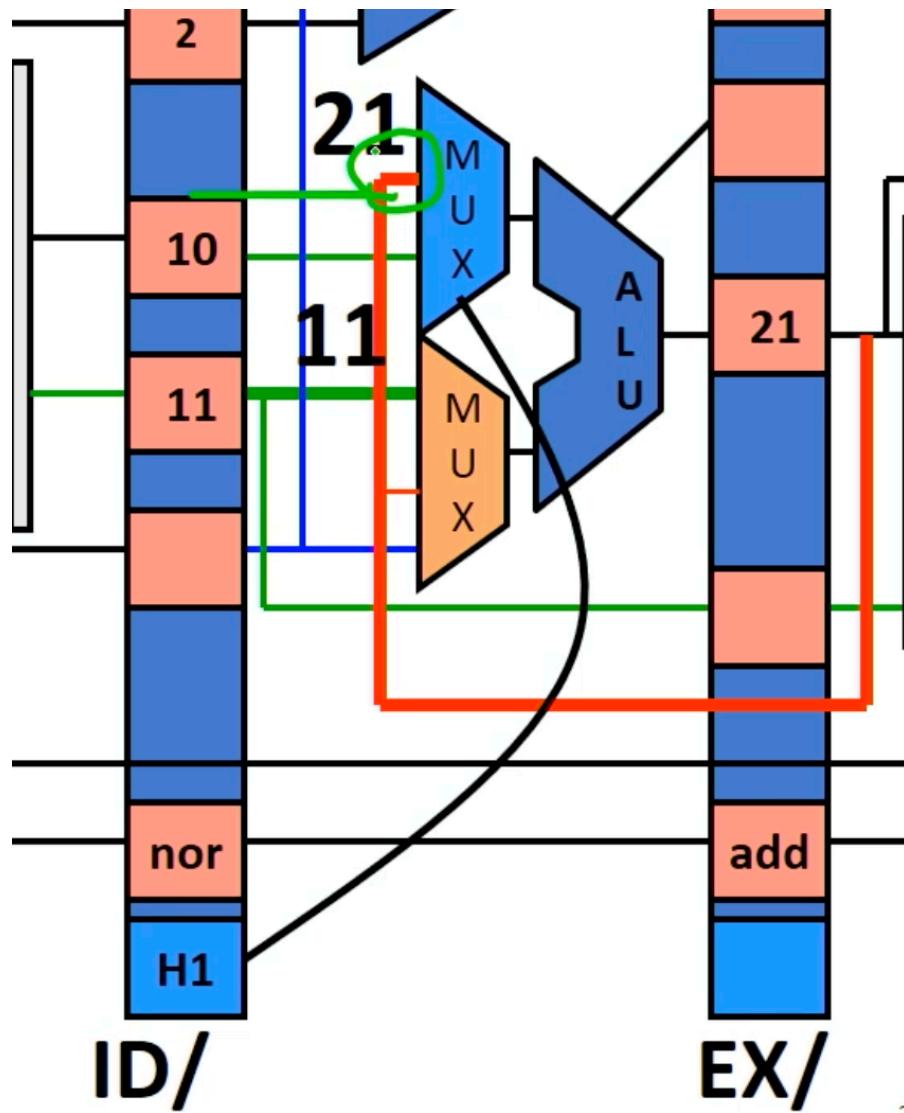
处理前四种 Data hazard

那么如何导数据呢？答案是，在 move 到下一个 stage 前，在通路上加上一个 mux. 根据我的 data hazard reg 的 encode (1/4), 去判断从下一个/下两个 stage 中的哪一个的 stage reg 上 grab value, 倒入到 regA/regB 上.

当当前 stage 的 data hazard reg 检测到 hazard 存在时，我们就把这个 mux 的 enable bits 设置为从对应的地方导入的数据 (下一个/下两个; regA/regB)

需要把下两个 Stage reg 的 regA/regB value 部分导入进来.





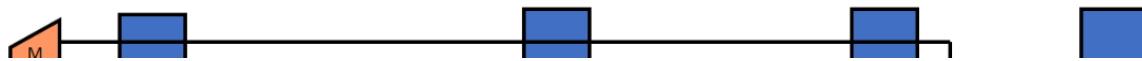
处理 sw/lw 的 data hazard

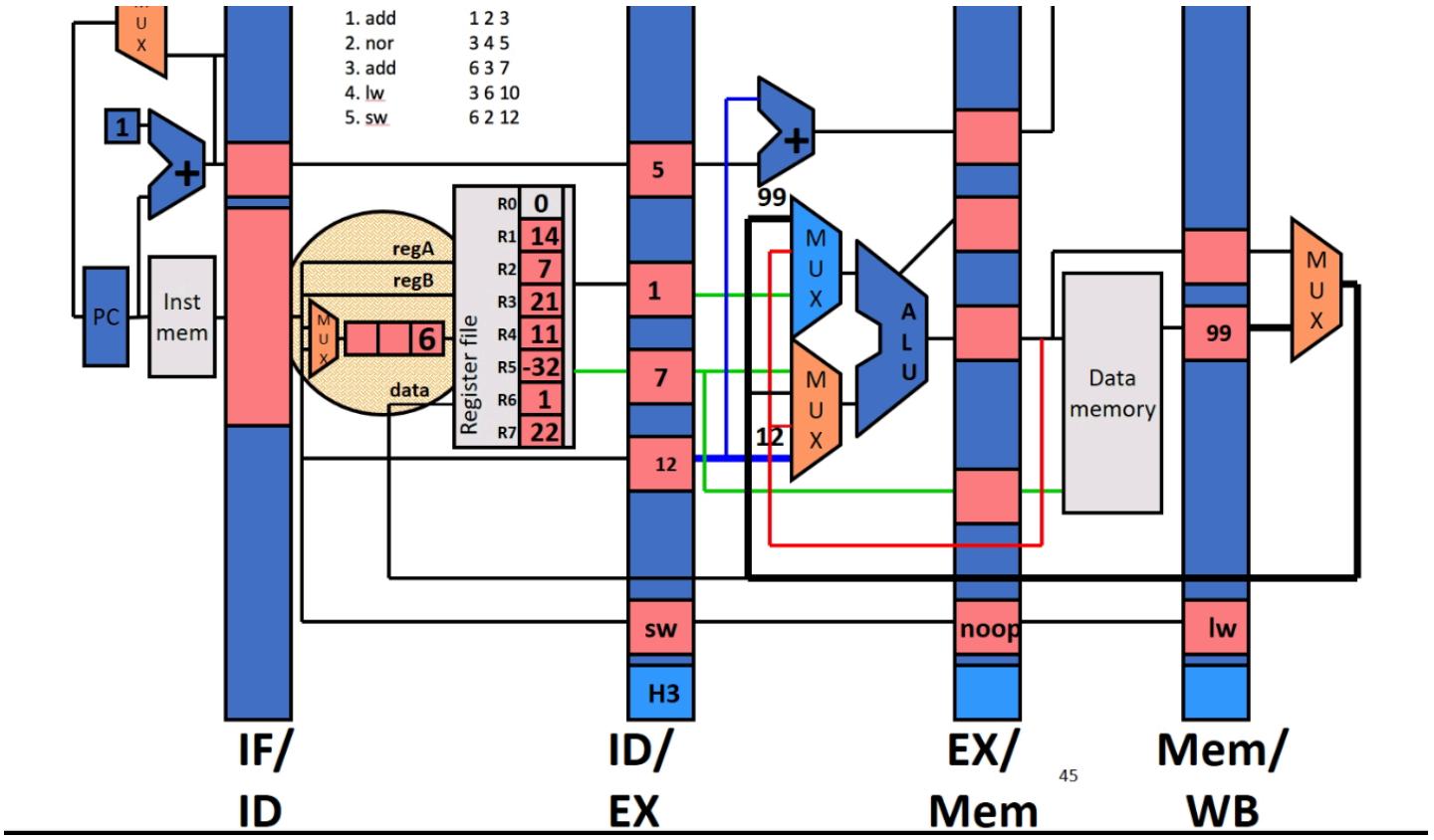
我们这里要结合 stall 的策略.

如果 lw 在 sw 前面两个位置，那么 sw stage3 的时候 lw 正好 stage5.

那我们直接把 write back 的结果导入一份到 stage 3 的 mux 就完事了.

First half of cycle 8



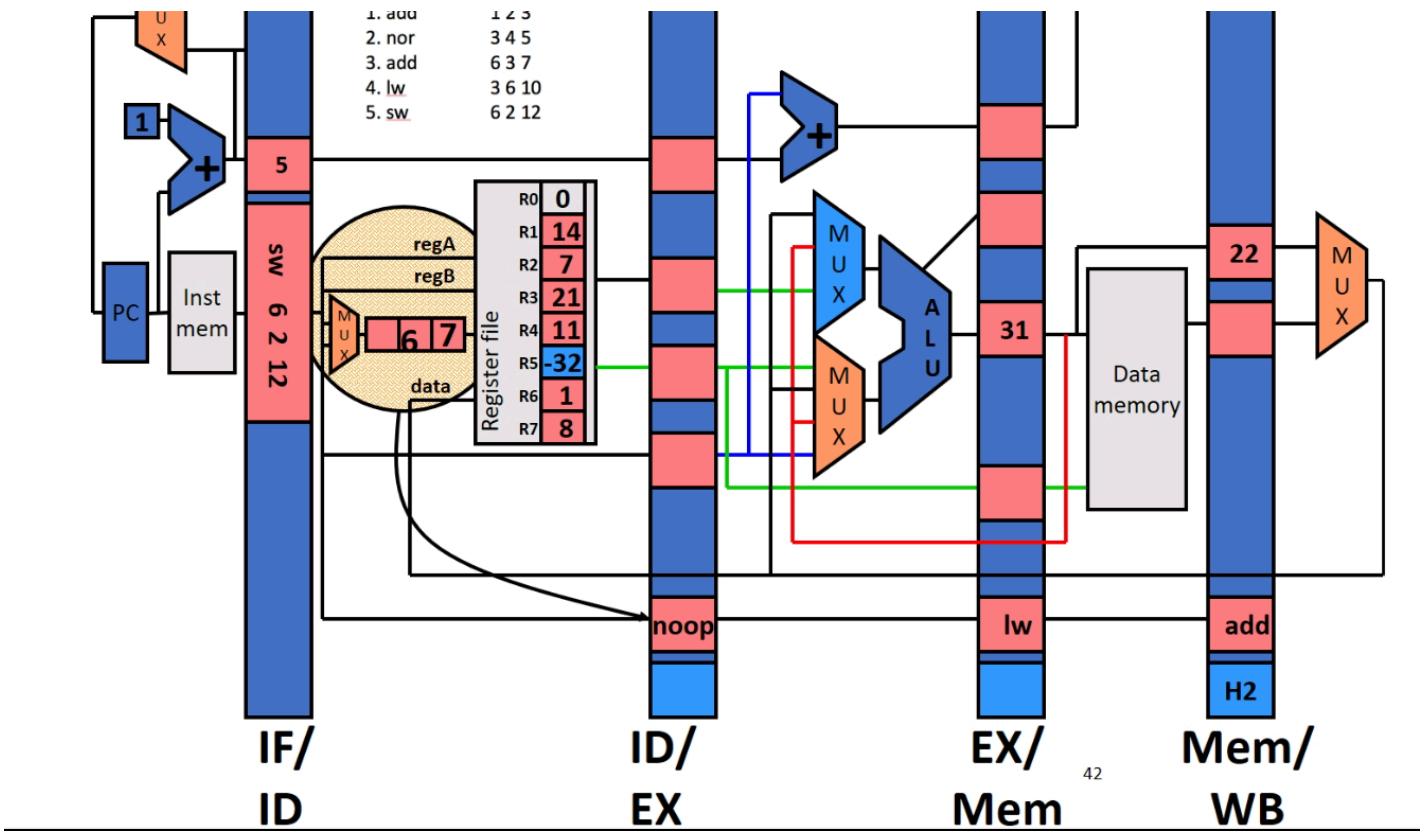


如果 lw 在 sw 前一个位置，那么我们必须在 Stage 1 detect 到 hazard 的时候就在前面插入一个 noop，强行把 lw 和 sw 隔开两个位置。

然后一样，sw stage3 的时候 lw 正好 stage5，直接把 write back 的结果导入一份到 stage 3 的 mux 就完事了。

End of cycle 6





42

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID	EX	ME	WB							
add 6 3 7			IF	ID	EX	ME	WB						
lw 3 6 10				IF	ID	EX	ME	WB					
sw 6 2 12					IF	ID*	EX	ME	WB				

Lec 15 Control Hazard

beq 的流程

Stage1: fetch inst

Stage2: read operands

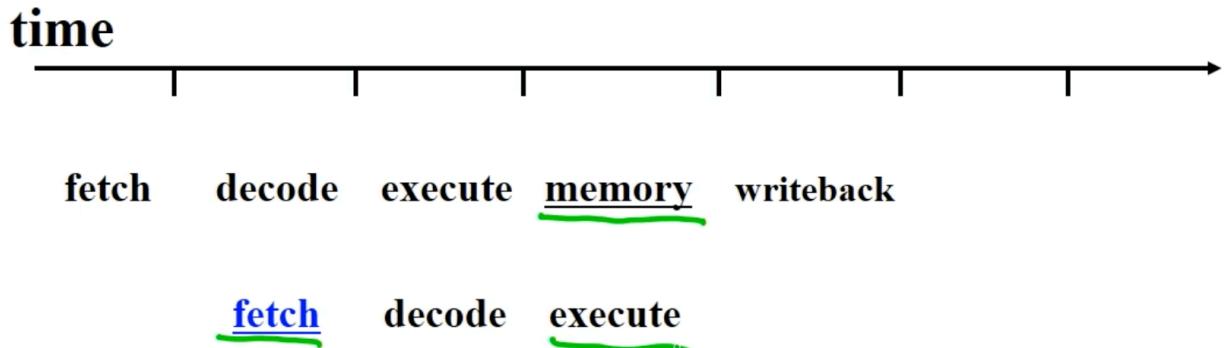
Stage3: 计算 target inst address; test for equality

Stage4: if equal, write PC = target inst address; otherwise PC ++

Stage5: 什么也不干

让我们 check 紧接 beq 的 instructions: 在 fetch 阶段就 fetch 错了. 后面全部都错了.

beq	1	1	10
add	3	4	5



Method 1: 在 assembly 层面避免 branch

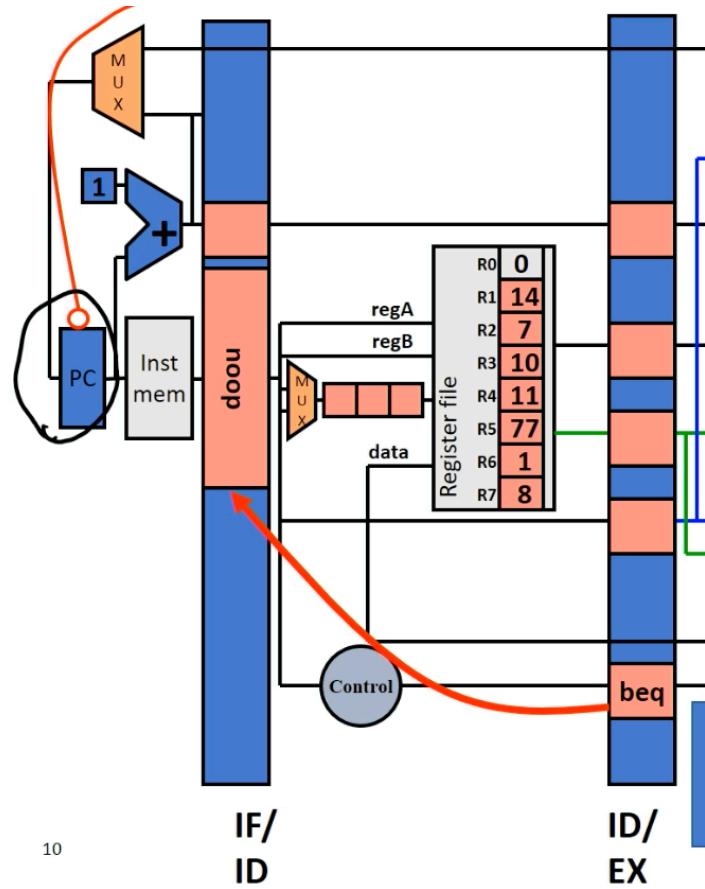
把 branch 全部写成顺序，把原本的函数体内所有 instructions 都换成 predicated instructions (比如 ARM 里的 conditions regs. 把 add 换成 add.pred)

但这只能在最基本的程序上作效。我们没法把每个 branch 都换掉

Method 2: Detect and stall

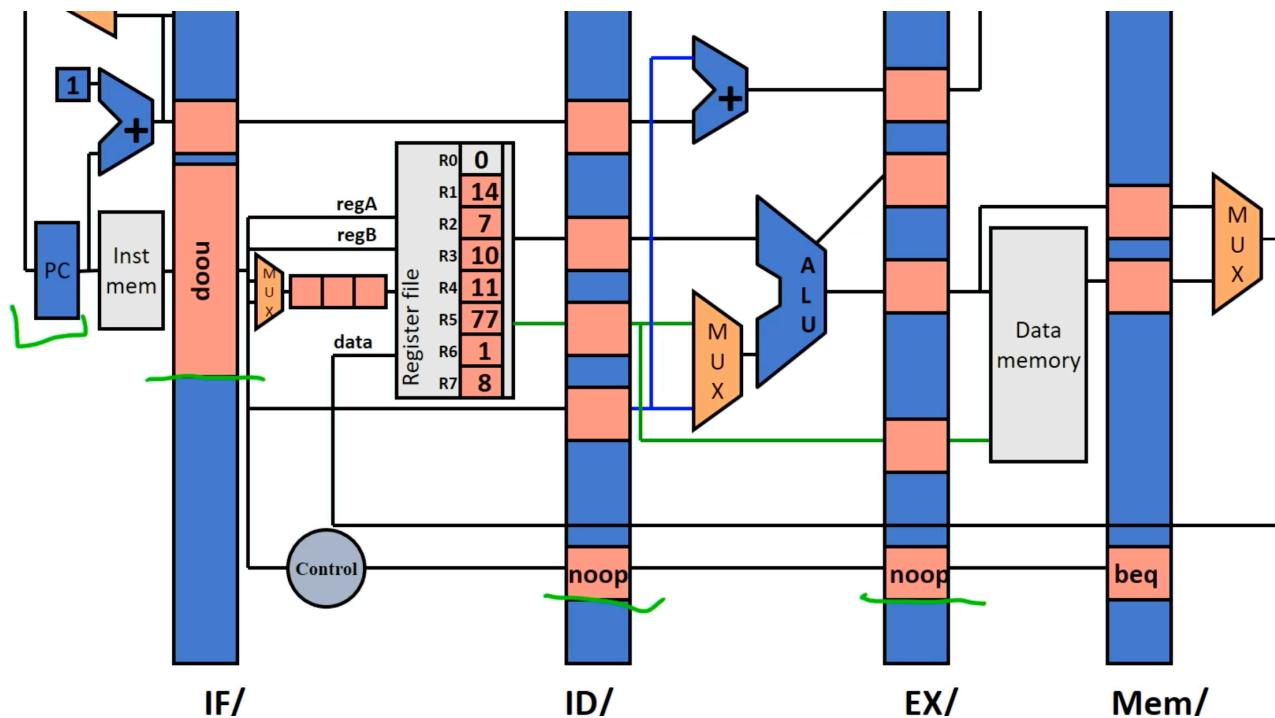
detect: 检测指令是不是 beq / jalr

stall: 一旦检测到 beq/jalr, 那么立刻把下一个指令保持在 fetch 阶段. 这样下一次还是 fetch 的同一个指令. 等于卡死在这里了



一直做这件事情三次，知道 beq 到 stage 5: 我们已经在 stage 4 结束的时候把 PC 的正确 address 传回去了，那么这次 fetch 的指令就是正确的指令了



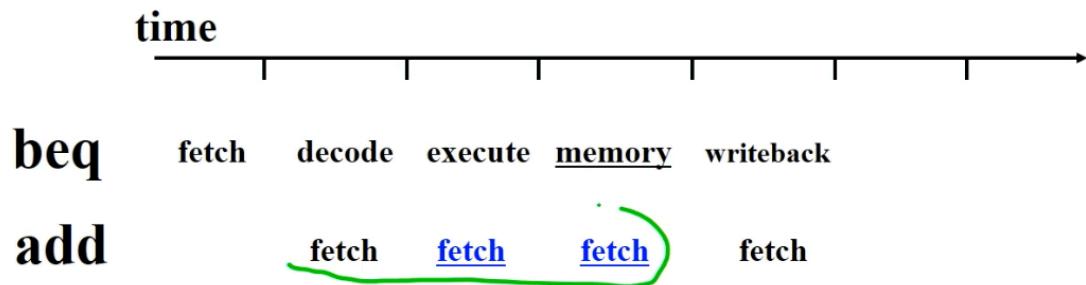


CPI 分析

beq 的 detect and stall 添加的指令是固定的：每次都添加三个 noop.

意味着每次遇到 beq， CPI 都只能增加.

beq	1	1	10
add	3	4	5



Method 3: Speculate and squash-if-wrong

想法：beq 只有在 equal 的时候运行，不 equal 就不会运行.

而 detect and stall 不论 beq 是否 equal，都会执行同样的 stall 策略。

这意味着我们可以优化策略：

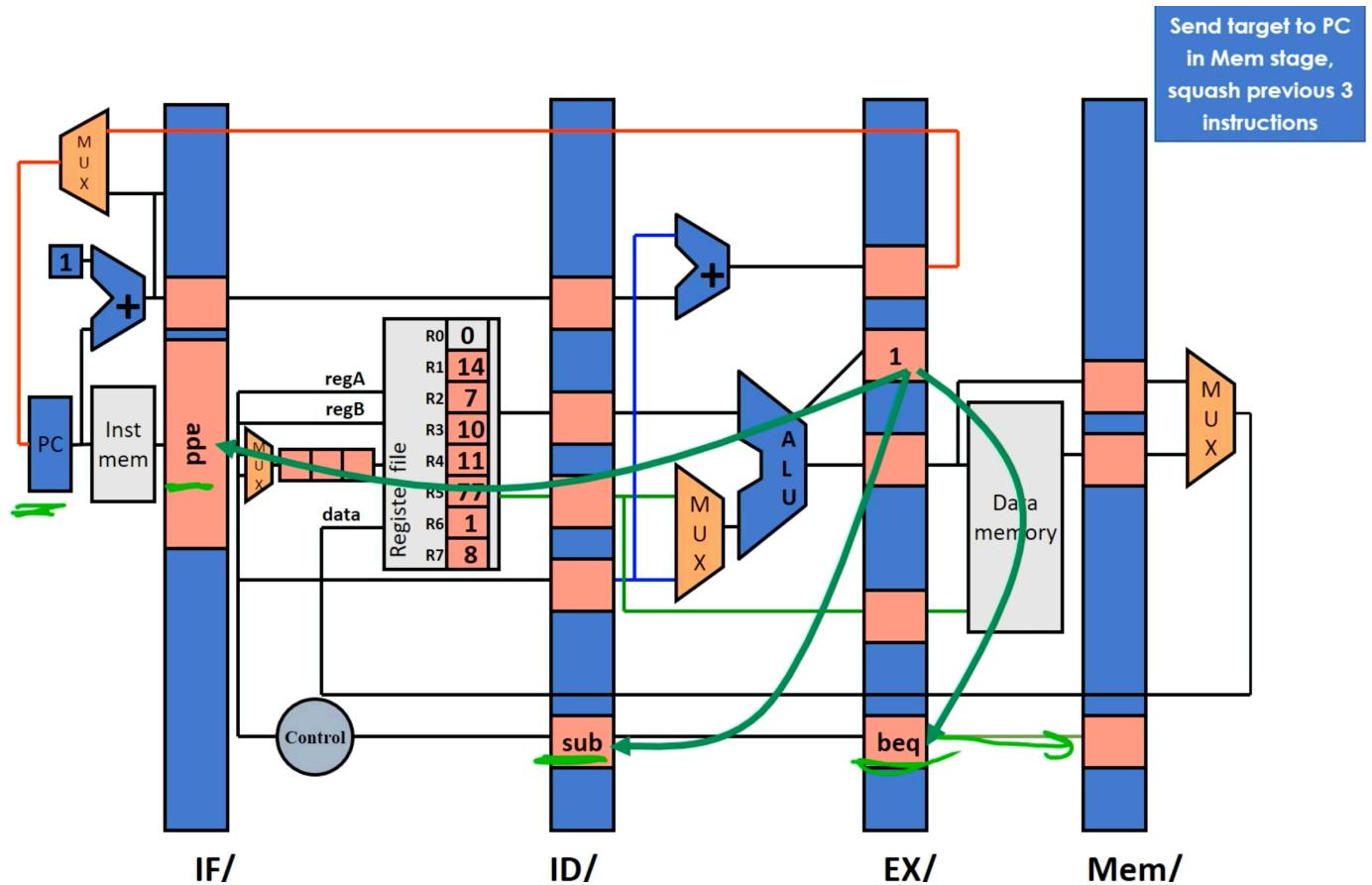
1. 假设不 equal，把 beq 当成 noop.
2. 如果第 Stage 3 发现 beq 是 equal 的，那么错误执行的两个指令如何处理？解决方案是我们不去完成这三个指令，而是一直把指令停止在它们的阶段，等待 beq 结束后的两个新指令把它们顶掉。

这个方法的可行点在于：前两个 Stage 不会 write back. 所以我们执行了错误的指令完全没关系。可以无痕覆盖掉。

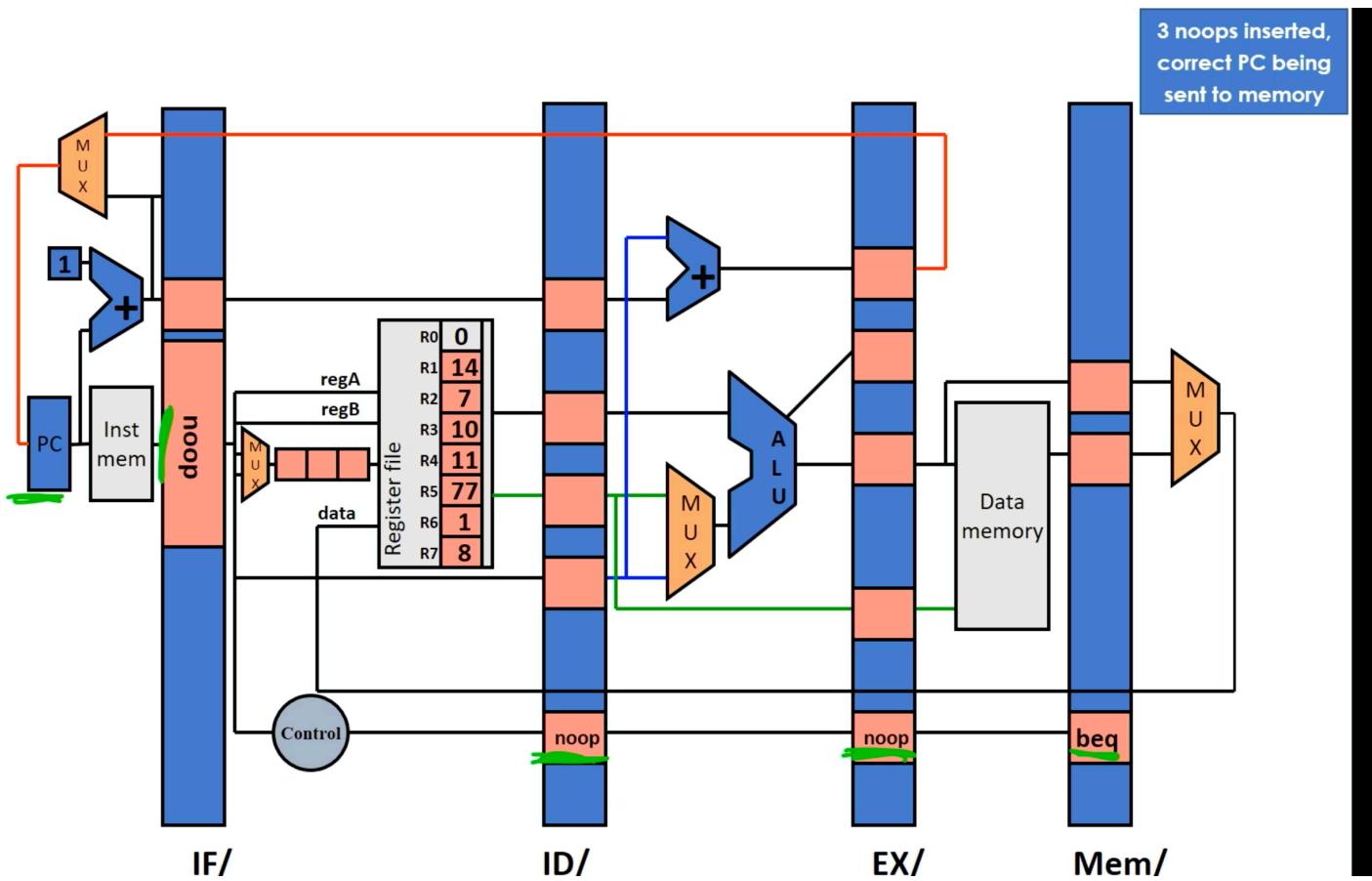
Squash 具体做法

当 Stage 3 检测到 beq equal 时，我们 send 三个 noops 分别到 decode, execute 和 memory 三个 stage，并把 ALU 结果（正确的地址）send 回 PC

于是下一步就当作无事发生，正常执行正确指令了



这个情况下的 performance 和 Method 2 的一模一样。不过因此它是一个绝对的优化，因为在 equal 时和 Method 2 一样，不 equal 时完全不增加 CPI，比 Method 2 好。



processor 一般总会使用这种方法.

除了 Data Hazard 和 Control Hazard 外的 Exceptions

a little bit digression: 当 something unexpected, other than data/control hazards 出现的时候怎么办? 比如说出现了一个 divide by zero

处理方法: 一个 ISA 被设计的时候会设计一些 "exception handler" 函数. 当出现意外情况就会 branch to 某个 exception handler 函数, squash 它之后的几个 instructions.

而平时的通常情况下, 这些 exceptions 的判断类似于 beq 的 "not taken", 不影响正常运行.

CPI Calculation Problems

Problem 1: 计算 CPI 和 TPI

lw	10%
sw	15%

beq	25%
R-type	50%

假设 20% 的 lw stall for 1 cycle

80% 的 beq not taken. 使用 speculate and squash 策略，其中 speculate 使用 "always not taken 假设"

$$\text{于是 CPI} = 1 + 0.1 \cdot 2 * 1 + 0.25 * 0.2 * 3 = 1.17$$

假设 clock frequency 100Mhz (于是1 cycle: 10ns)

$$\text{那么 Time per instruction: } 1.17 * 10\text{ns} = 11.7 \text{ ns}$$

$$(\text{Time per instruction} = \text{CPI} * \text{Time/Cycle})$$

Problem 2: Adding a PC Writeback

试想：如果我们在 Execute Stage 额外加一次和 Mem op stage 一样的 PC write back，那么我们就可以只插入两个 noops 而不是三个，以 resolve control hazard 了.

于是 CPI 将减少：

$$\text{CPI} = 1 + 0.1 \cdot 2 * 1 + 0.25 * 0.2 * 2 = 1.12$$

但是，我们关心的其实不是 CPI 而是 TPI.

Question: Will TPI be decreased?

Answer: Depends on the cost of adding the MUX&Writeback. 因为这可能会增加一个 clock 的时间。

Note: 一个 clock 的时间是所有 5 个 Stage 中最长的一个 Stage 的时间！ 如果我们增加了这个行为后，Stage 3 的运行时间变得比原本的五个 Stage 中最长运行时间的 Stage 更长了，那么我们就不得不增加 clock period 即 Time/Cycle，也就是减少 clock frequency. 这样所有的 instructions 执行时间都变慢了！

最后是否 TPI 更小需要看 CPI 的减少和 Time/Cycle 的增加谁更显著.

Problem 3: 10 stage pipeline

现在我们新引入一个 pipeline 架构：10 Stages

Instructions are fetched at stage 1.

Register file is read at stage 3.

Execution begins at stage 5.

Branches are resolved at stage 7.

Memory access is complete in stage 9.

我们于是要抓住解决 hazard 的核心：

解决 hazard 的逻辑总结

Data Detect and stall: 把新指令留在 decode stage, **impose noop** 直到 **dependent** 指令的下一个 Stage 是 **writeback stage** 的时候 **move** 它.

Data Detect and forward: 灵活检测导 reg. 只有 lw/sw 需要处理 stall. 当出现 sw 需要 lw 存进的 reg 时, 插入 **noop** 确保 **lw** 到 **writeback stage** 的时候 **sw** 运行 **execute** 就可以了

Control Detect and stall: 把 **beq** 的下个指令留在 **fetch** 直到 **PC** 传回的 Stage 结束之后再正常 **fetch**.

Control Speculate and squash-if-wrong: 在 **beq** 的 **eq** 判断前一切正常, 判断后如果是 **true** 则把从 **decode Stage** 一直到 **beq** 的下一个 Stage reg 上都 **insert noop**.

于是 $CPI = 1 + 0.1 * 0.2 * (9 - 5) + 0.25 * 0.2 * (7 - 1) = 1.38\$$

pipeline 的增加会伴随 clock period 减少。假设 clock period 变成了原来的一半, 那么 TPI 将大大增加.

Branch Prediction: 更多 Speculation 策略

需要 predict 哪些东西

speculation 即猜测。但是其实我们上面只讲了一种策略：猜测总是 beq 总是不 equal.

我们也可以 make 其他猜测。这就是 branch prediction

branch prediction 的目的：尽可能地减少 noop 插入的次数，从而降低 CPI.

我们目前的 branch prediction 策略：beq 总是不 equal, 是一个最简单的策略。它的结果是一旦 equal 则必然插入三个 noops。这显然不是最优的。

我们需要的是：在 branch 结果是 equal 的时候也可能顺利进行，不去 squash 的策略。

意思是：当我们预测一个指令是 branch 的时候，我们直接预测并跳到它的 target address 上，最后如果不 take 就 squash 并回归原来的正常的指令。

为此我们需要 predict 的东西有：

1. 某个 **instruction** 是否是 **branch**
2. 如果是 **conditional branch** 的话, **condition** 结果 (比如我们的 **beq**: 是否 **equal**?)

3. 如果 take 了这个 branch, 那么 branch address 是什么

(看到这里我的感觉：很显然如果对是否 branch 的预测影响到正常其他指令的运行就得不偿失了，所以这里所谓的 "预测是否是 branch" 一定只是储存已经遇到过的 branch，当再次遇到的时候直接模仿上一次的行为罢了。所以实际上根本不是预测而是 remember old route.)

Predict branch & branch address

Idea: 记得遇到过的 branches.

注意到：同一个 branch 的 Target address 总是相同的。所以当我们预测一个 instruction 是 branch 的时候，我们同时预测了 3.

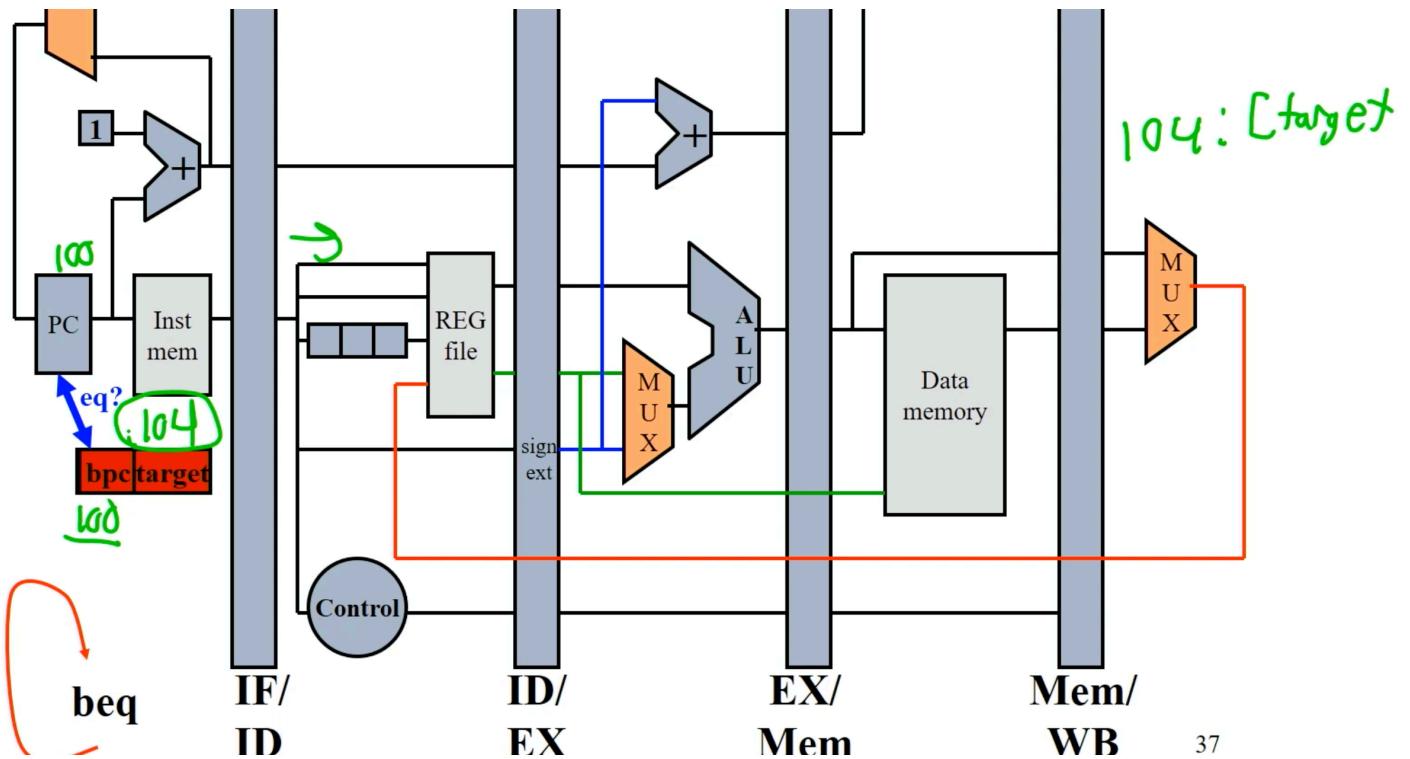
所以我们的 idea: 每次我们执行 branch 就 store 当前 PC 的地址，以及它的 target address.

看起来没什么用，但是其实很有用。因为这样就很好地处理了循环：试想在一个循环里，我们除了最后一次 loop 之外，其他每次 loop 都会 take branch 并且 branch 到相同的地方。

具体实现方法：当我们遇到一个 branch 时，我们记住它和它的 target address.

每次在 fetch 的时候，我们都留意当前的 PC 地址：如果我们记住的 branch instruction 的地址，我们就根据 branch direction 的 predict 结果来决定是否直接 branch 到 branch address.





37

所以整个逻辑就是这样的：

在 fetch stage, 我们查看 target addresses 的 cache, 看看我们当前的 PC 在 +1 后是否是某个 target address
如果不是则正常运行.

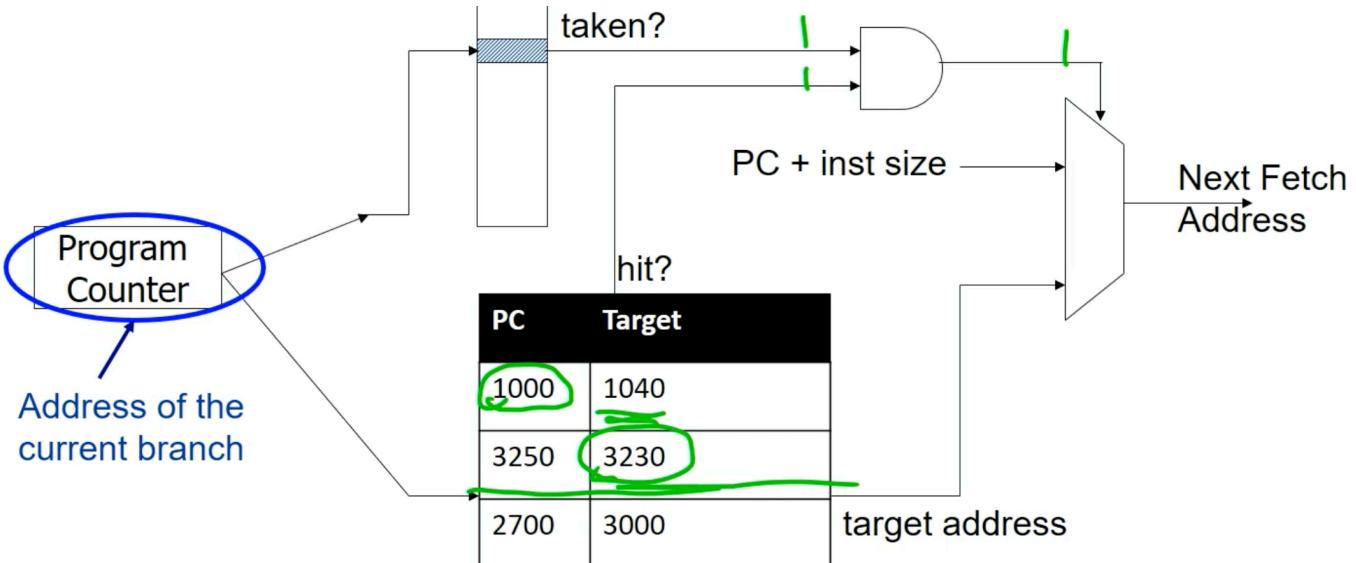
如果是的话, 我们再查看 direction predictor: 这次 branch 是否选择应预测为 take.

如果 take: 直接 fetch target address

如果不 take: 正常运行.

Sometimes predict taken?

Direction predictor



"Cache" of Target Addresses (BTB: Branch Target Buffer)

predict branch direction

我们还缺少最后一个组件: predict 是否应该 take 某次 branch

有两种主要的类别: static/dynamic method

Static 表示 predict once during compilation, execution 时总是 predict 相同结果 (比如我们最开始的方案: 总是 predict not branch)

Dynamic 表示在 execution 时 prediction change over time.

Dynamic VS Static strategies 是 Computer Architecture 的一个常见的 topic

Static branch prediction

1. always predict taken
2. always predict not taken
3. **backward taken, forward not taken (BTFN)**

根据实践经验: 大概 60% 的 conditional branch (in LC2K, beq) 都是会被 taken 的. 所以 always predict taken 在概率上优于 always not taken

BTFN 是一种不错的 static 策略, 因为 Backward branch 通常是 loop, 而我们知道 loop 除了末尾都会 take branch. 所以准确率应该不错

Dynamic branch prediction

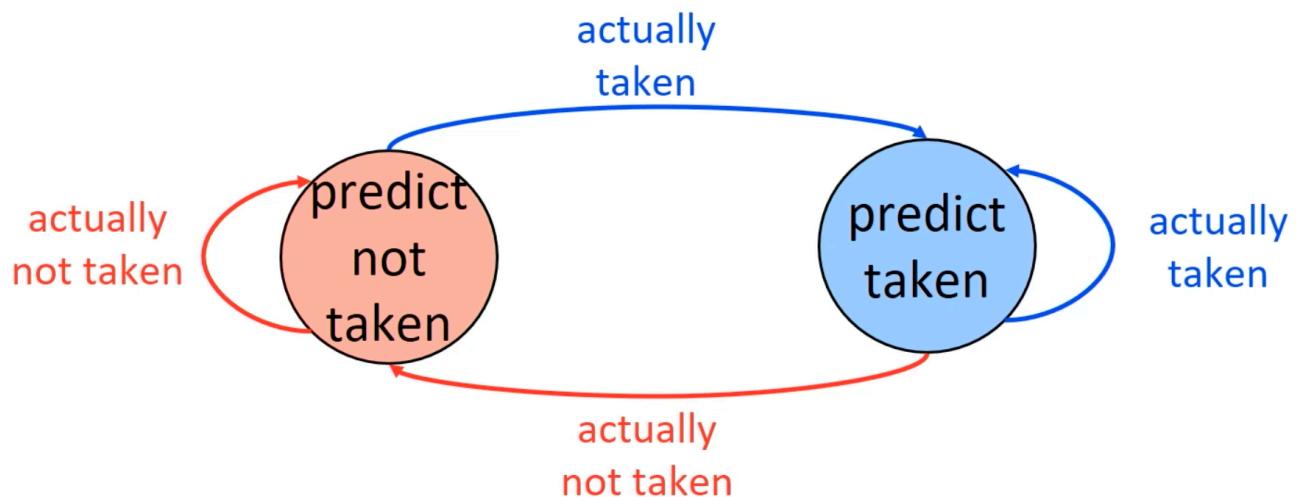
1. last time predictor: 最简单的 dynamic branch prediction. 总是给出上一次 branch 是否 taken 的 execution 结果

example: TTTTTTNNNNNN, 90%的准确率

always mispredict loop 的第一个和最后一个 branch ($N-2/N$ 的准确率 for loop)

0% 准确率 for TNTNTNTNTNTN....

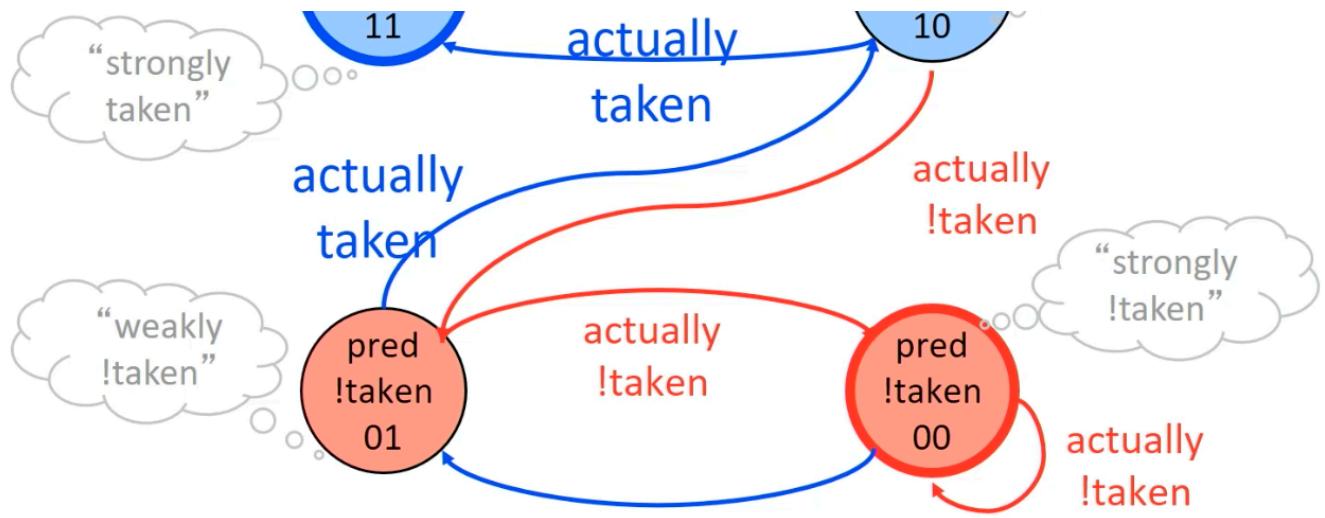
State Machine for Last-Time Prediction



2. 改进 last time predictor:

只有在连续两次错误后才更改 pred





Ex:

Br	T	T	T	T	N	T	T	N	N	N
State	10	11	11	11	11	10	11	11	10	01
Pred	T	T	T	T	T	T	T	T	T	N

实际上,

pred not taken: ~50% accuracy

pred taken: ~60% accuracy

last time predictor: ~80% accuracy

realistic predictor(使用ml, ml system): ~96% accuracy

Note Part 3: Cache and Memory Design

Lec 16 - Cache

By complexity 左右两边是一样的. 但是考虑到实际的 cache, 左边的代码远比右边快

Cache Aware vs Non-Aware Code

```
#include<stdio.h>
#include<stdlib.h>

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
    int i, j;
    int count = 0;

    for(i=0; i< N; i++)
        for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[i][j] = 10;
    }

    printf("Count :%d\n", count);
}
```

```
#include<stdio.h>
#include<stdlib.h>

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
    int i, j;
    int count = 0;

    for(i=0; i< N; i++)
        for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[j][i] = 10;
    }

    printf("Count :%d\n", count);
}
```

目前我们 discuss 了两种 hold data 的 structures: reg file; memory

Reg file 和 memory 都是 array of words.

其中每个 word 都是一堆 32 个 bits(for LC2K, 64 for ARM) 组成.

所以我们储存数据的最底层结构是：我们如何储存一个 bit?

根据之前学的，一个 bit 可以由一个 flip-flop 来表示

但是这显然不 practical 因为一个 flip-flop 里面的 transistor 太多了，而我们需要 $2^{18/32/64}$ 个这样的单位.

Options to represent a bit

reg files, 数量极少 (就几十个 regs)

由最贵最大的 D flip-flops 组成.

其他的 memory: 有下面几个由快到慢的 options

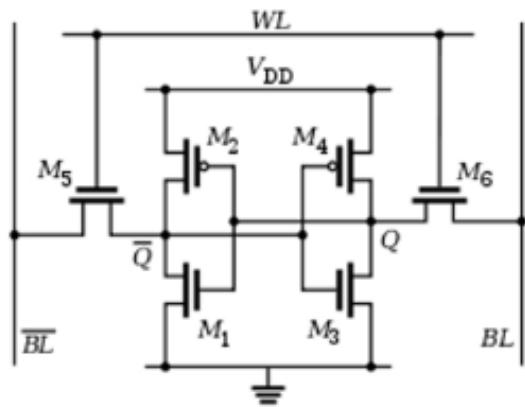
Option 1: SRAM(Static Random Access Memory)

用 SRAM 来表示的一个 bit 由 6 个 transistors 组成

它的优点是很快：1 ns read time，只慢于 flip flop.

虽然显著比 D flip-flop 小，但它仍然很大。我们可以在一个 chip 上放 ~MB 个这样的东西；但是 ~GB 级别的 memory array 不可以用它表示

当然它需要 constant power 来 keep data.



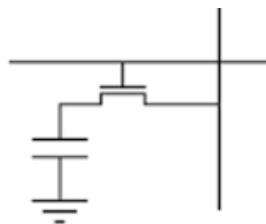
主要用于：caches.

Option 2: DARM(Dynamic Ramdom Access Memory)

用 DRAM 来表示的一个 bit 由 1 个 transistor 和 1 个 capacitor (电容器) 组成

transistor 负责电荷进出，capacitor 负责储存电荷。

capacitor 的电荷会随着时间逐渐流失，因而需要周期性刷新来保持电荷。所以它也是需要通电才能储存，断电后数据会流失。



慢于 SRAM. 50 ns read time (因为它使用了 capcaciator, refill 需要时间. 不像 SRAM 是纯粹靠 transistor 来维持 0/1 的.)

比 SRAM 更便宜. (transistor 是耗费大头，这里只有一个 transistor)

可以在 current machine 上放 ~16-64 GB 个。

对于 32 bit systems 是好的选择，但是对于 64 bit systems 则不是。

主要用于: main memory 以及显存等

Option 3: Disks

Disks, 古早的是机械硬盘, 使用 megnetic charges 来储存数据. 它的储存介质是通过 spinning disks 来完成.

HDD(Hard Disk Drive)

disk 的盘片表面有磁性材料, 通过磁性的改变来表示 0/1

读写磁头由机械臂控制 (所以叫机械硬盘), 通过移动它到盘片的不同位置来进行数据读写.

disk 上有spinning 的电机, 带动磁盘高速旋转, 使得磁头可以快速定位到需要的数据上 (这个"快速"相对于 DRAM 是非常非常非常非常慢的

速度: 3000000 ns access time. 慢.

SSD(Solid State Drive)

固态硬盘. 无机械部件. 通过 NAND 闪存来存储数据.

机制比较复杂。使用浮动栅极晶体管, 能够锁定电荷, 用电荷是否存在表示 0/1.

寿命也有限。所以硬盘老了得换。

现在应该没人用机械硬盘了. SSD 的储存范围更大, 可以 scale up to terabytes, 单个 MB 的价格只需要 0.0001 刀.

SSD 的速度比 HDD 快很多. 但是仍然远远慢于 DRAM.

Memory Hierarchy

我们想要一个尽量 fast (Ideally run at processor clock speed) 且 less expensive 的 memory system.

DRAM 和 Disks 太慢, SRAM 太贵/大.

所以我们需要 memory hierarchy.

Idea: 对于 main memory, 介于大小的需求(GB级别), 我们只能使用 DRAM. 但是, 我们实际上在一段时间内只需要 **access a small amount of data** 而不是整个 memory.

所以: 我们可以用 a small array of SRAM to hold data we need now. 这个 small array of SRAM 就叫做 cache (缓存).

所以最后我们的 hierarchy 是：

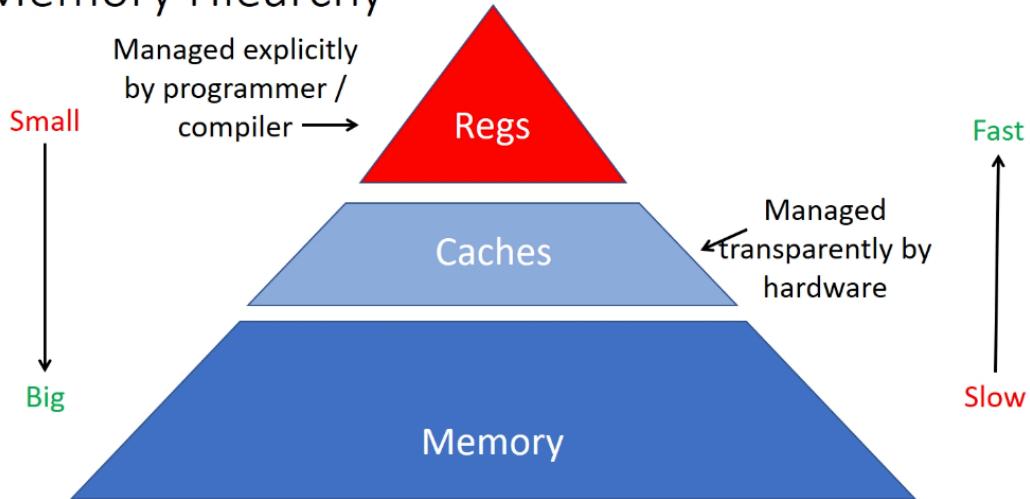
regfiles: D flip-flop

~Kb 的 cache, 由 SRAM 组成

~Gb 的 memory, 由 DRAM 组成

Everything else, stored on disk (现在有 Virtual memory, 允许计算机使用 disk 作为扩展内存, 以在 RAM 不足时提供更大的可用内存空间。VM 的主要目的是让程序认为自己有足够的连续内存, 即使实际的物理内存可能不够, 从而增强系统的多任务处理能力和程序运行的稳定性。)

Memory Hierarchy



一个 analogy: memory, cache, reg

Cache Analogy

- Studying in the library

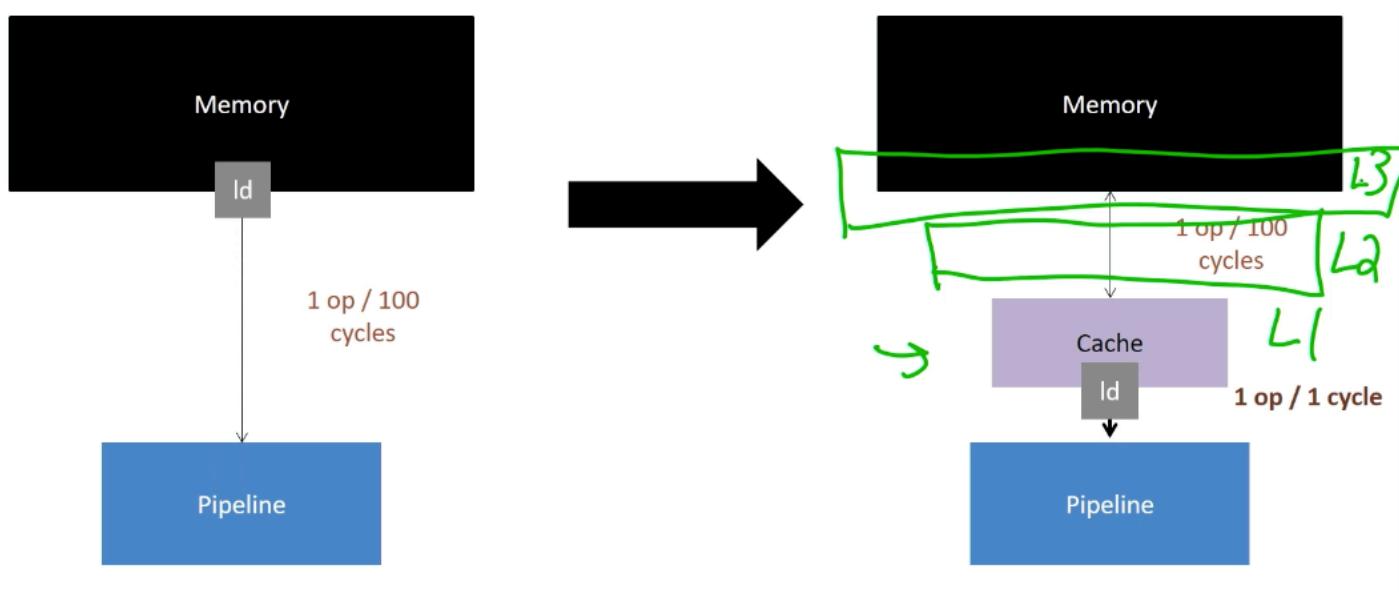
Summary of the library

- Option 1: Every time you grab another book, return current book to shelf and get a new book from shelf
 - Latency = 5 minutes
- Option 2: Keep 10 commonly-used books on shelf above desk
 - Latency = 1 minute
- Option 3: Keep 3 books open on different locations on desk
 - Latency = 10 seconds

Function of the Cache

我们使用 cache 来把 memory 中常用的数据放进其中，这部分数据之后在 cache 里和 pipeline processor 之间进行交互，而不是在 memory 里和 pipeline processor 之间进行交互，大大提高了速度。

实际上 Cache 也分为 L1, L2, L3 级别，和 cpu core 距离由近到远，大小由小到大，速度由快到慢。越常用的数据和指令放在越靠前的 cache.



Cache holds the data we think is most likely to be referenced.

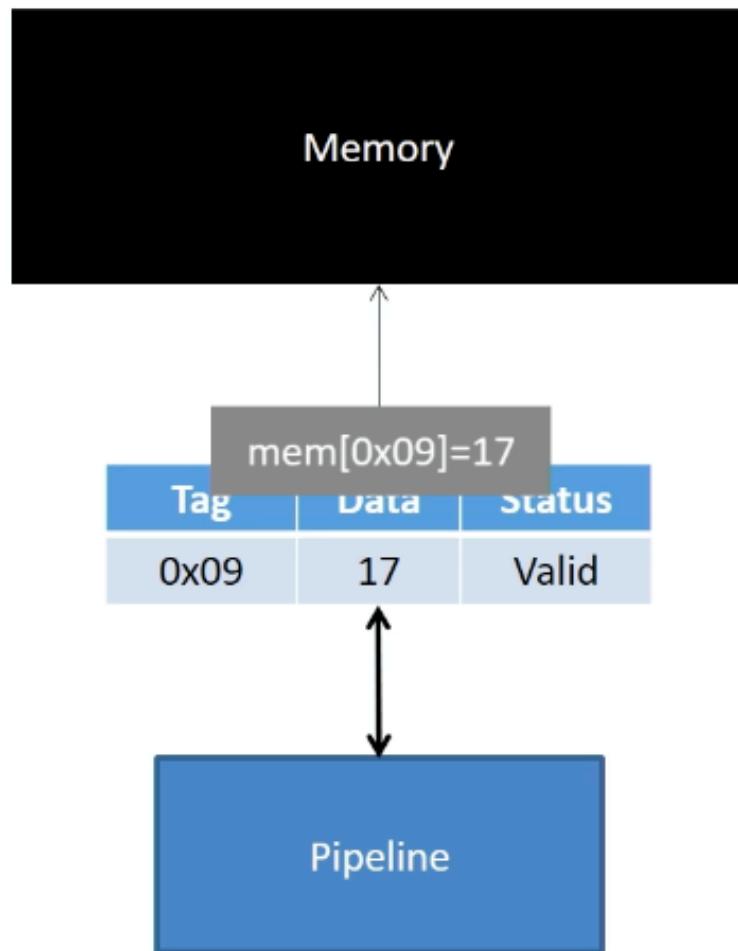
我们首先考虑 LC2K 中最简单的 cache: 只用于 load instructions 的 cache:

这个 cache 是一个 word addressable address space.

当我们 lw 时，memory 会 return 一个 data。我们

1. 把 memory return 的 data 存进 cache,
2. 同时 together with 一个 tag: 储存这个 data 在 memory 中的地址, 以便下次查找
3. 同时 together with 一个 one bit status: 表示这个data 是否 valid.

由这几个东西构成一个 word. 作为 cache 中一个位置上的 word.



temporal locality

我们在 cache 中储存的时候需要考虑这些问题

1. cache 是否足够大, 以每次遇到需要储存的东西都能够有空储存?
2. 如何选择 what to keep in cache?

我们有一个 locality 的原则: 程序具有 temporal locality 和 spatial locality

temporal locality 即：如果一个 given memory location is referenced now, it will probably be used again in the near future.

(显然的，因为我们写代码肯定会用一个 variable multiple times)

并且由此可 corollary：如果我们很长时间不用一个 variable 了，说明它很可能任务完成了（之后不会再出现

cache 可以利用这个特性：

1. 对于 just accessed 的 data 应该 place into cache
2. 当我们必须要 evict something 的时候，我们 evict whatever data was **Least Recently Used(LRU)**

example

Example: 我们采用一个简化的模型. 这里 Cache 的容量一共就 2 entries, 每个 line 由 1 bit status, 4 bit tag, 8 bit data 组成 (没有 LRU bit, 因为一共只有两个 entries, 一个被访问了之后另外一个就是 LRU data)

这里的 memory 是一个 bit addressable space, 一共有 16 lines,

地址是 0b0000 - 0b1111

最大是 4 位，所以我们直接采用 tag = address, 使用 4 位的 tag 来标记

Cache overhead

overhead 指的是 cache 中 non-data 的部分

这里，我的 data 一共 $2^2 \times 8$ bits = 16 bits 16 bits

而 overhead:

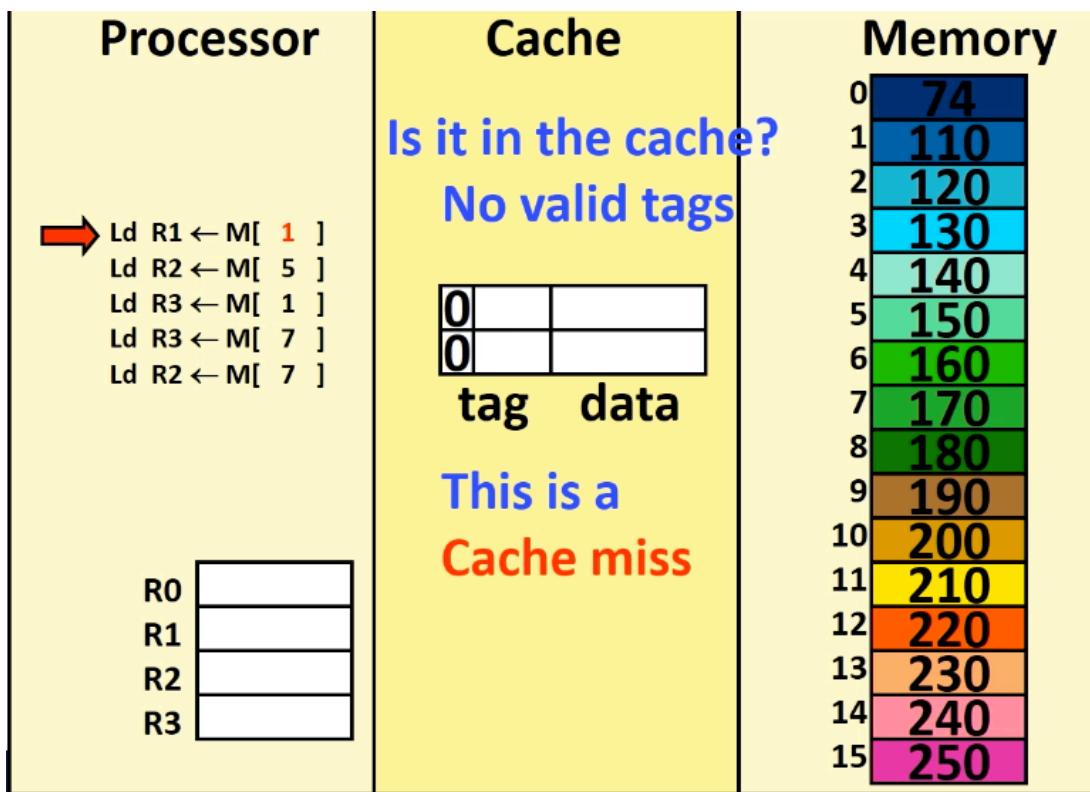
1. tag $2^2 \times 4$ bits
2. valid bits $2^2 \times 1$

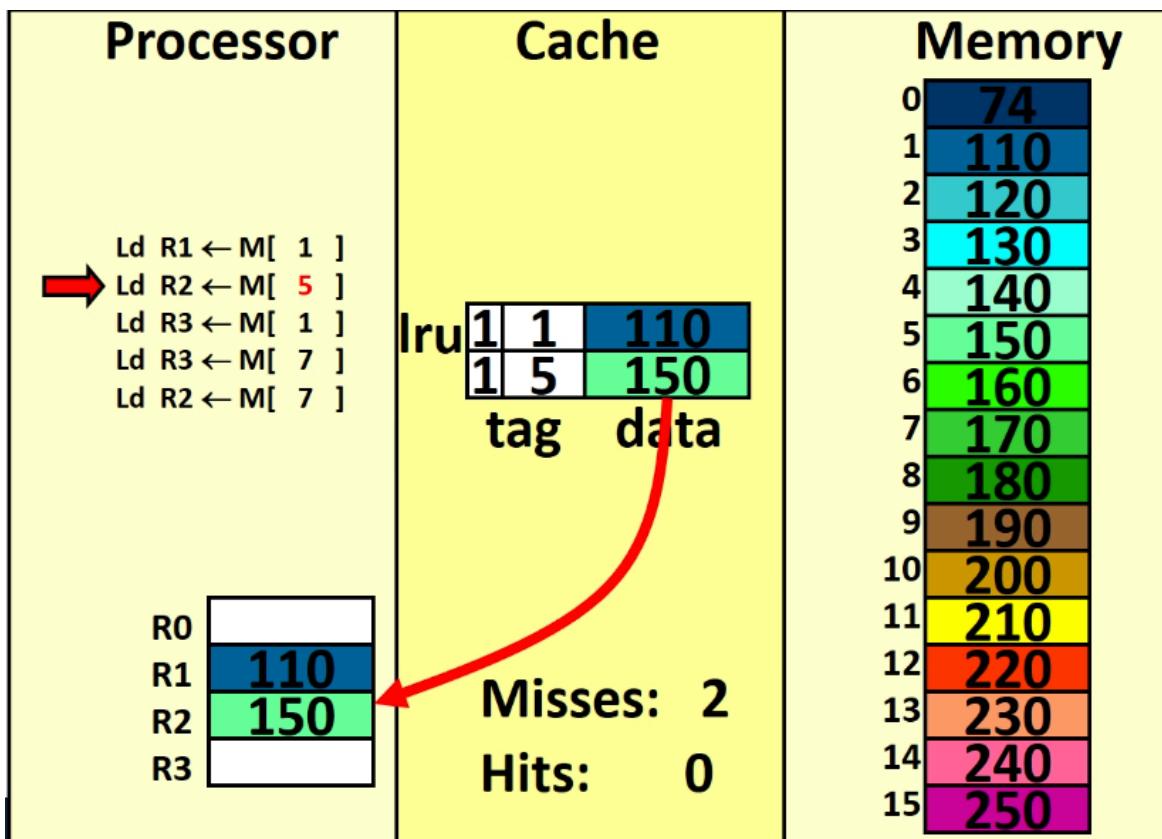
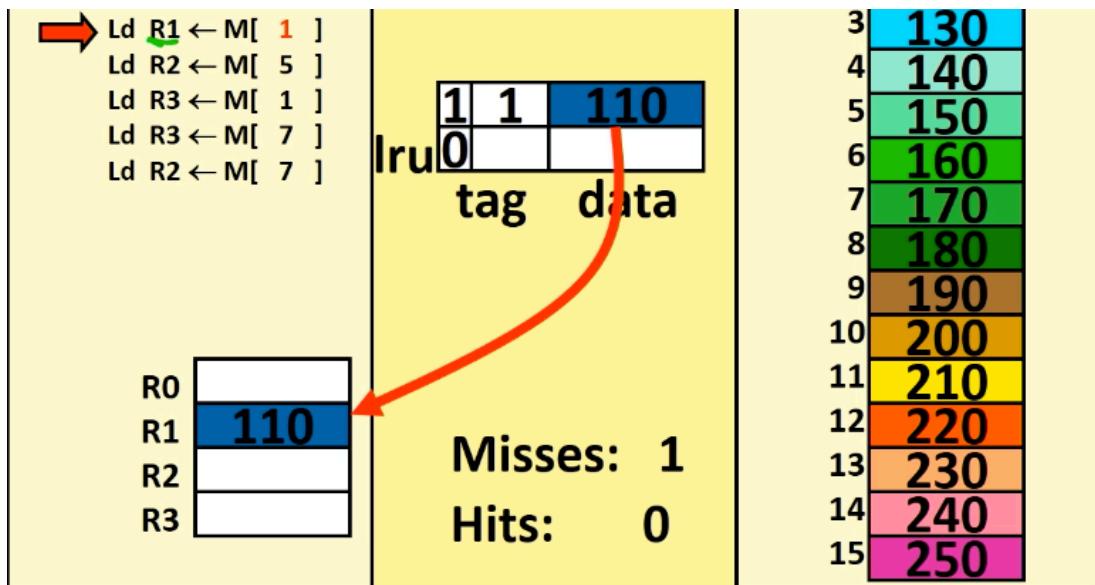
一共 10 bits 的 overhead

每当 lw 时，我们首先遍历 cache，找不到的话：

1. 记录一个 Cache miss;
2. 在 memory 找，并且把找到的数据的位置作为 tag，内容作为 data 放进 Cache 的 LRU 的地方，Status 设置为 1

3. 从 Cache 中数据进入 reg.

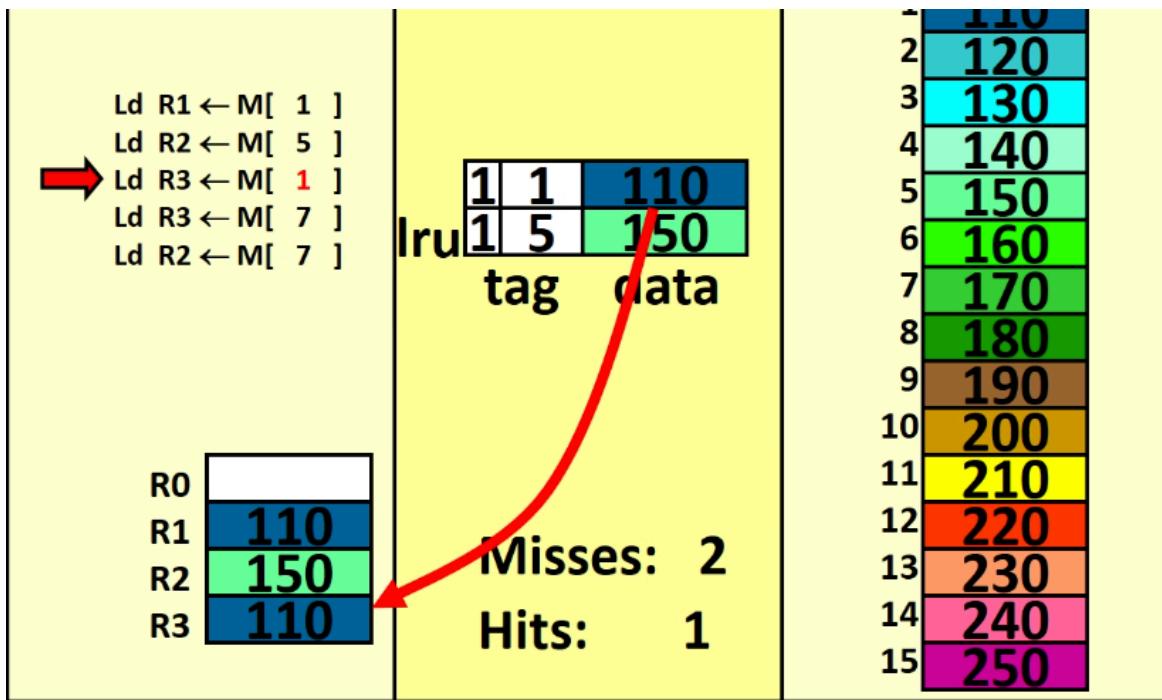




遍历这块 Cache, 找到的话:

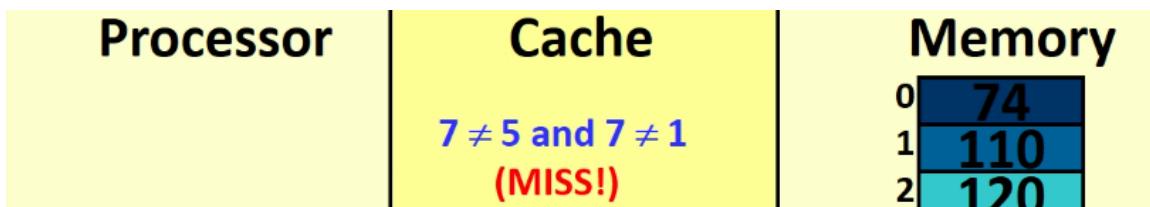
1. 记录一个 Cache hit
2. 直接从 Cache 里把数据导进 reg

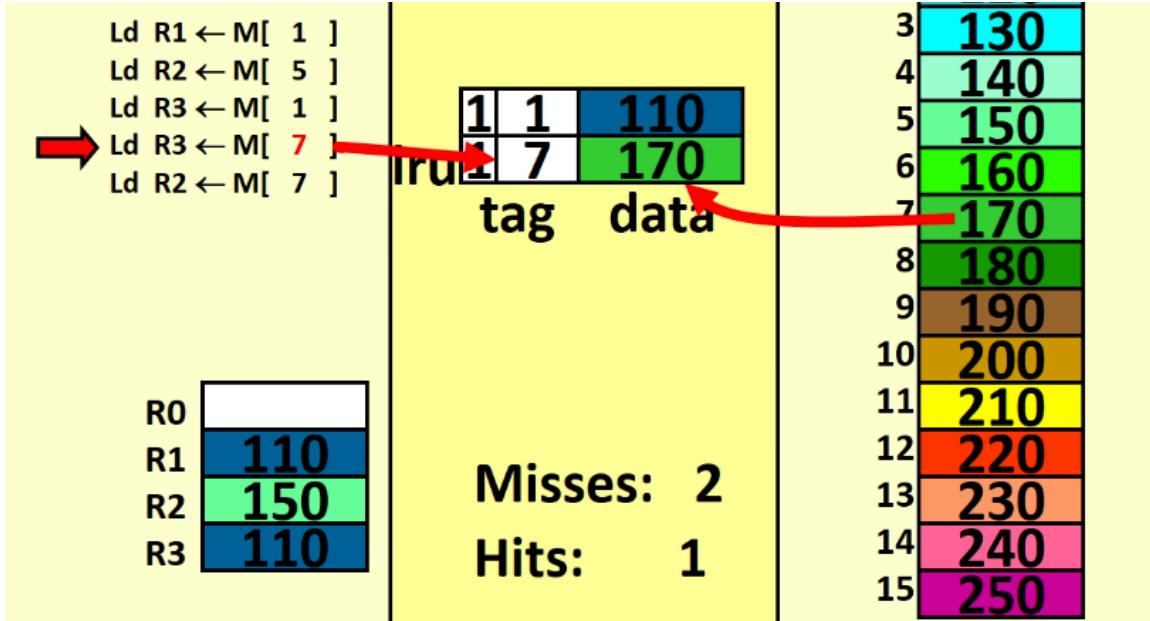




如果 Cache filled

当 Cache 被填满的时候，我们如果遇到一个新的 Cache miss，意味着又要放数据进来。我们只能 kick out 一个相对 LRU 的数据





这一块的逻辑应该是：这里的模型过于简单，实际上 status bit 会不止一个，最后从头到尾遍历比较 status bits 来找出最 LRU 的数据

这一部分在 next lecture

Hit/Miss rate

Hit 指 data for a memory access 在 cache 中被找到

Miss 指 data for a memory access 在 cache 中没被找到

我们想要一个比较高的 Hit/Miss rate

Ex:

假设 Cache 的 access time 是 1 cycle

Main Memory 的 access time 是 100 cycles.

如果我们有 90% 的 Hit/Miss rate，那么 average memory latency 是多少？

$$1 * 0.9 + (1 + 100) * 0.1 = 11.0$$

使用公式：

$$\text{average latency} = \text{cache latency} + \text{memory latency} * \text{miss rate}$$

$$1 + 100 * 0.1 = 11.0$$

因而要优化 average latency, 我们可以选择

1. 优化 memory latency
2. 优化 cache latency
3. 降低 miss rate

Lec 17 - Improving Caches

原本的 cache 模型: 4 个 tag bit, 直接使用地址作为 tag

问题: 一共只容纳了两个数据, 太容易 miss 了。例如: 一个循环, 里面有两个 local variable, 于是每个 iteration 之后, iterator 变量就被 cache 挤出去了。所以这个 cache 基本等于没有。实际上一个循环, local variable 有两个是很正常的。只能说明这个 cache 太弱了。

我们想要在能够寻址整个 memory 的同时, 还能够尽量地多存储一些数据放进 cache 里。

办法: 我们直接把 cache 做得更大点, 容纳更多的数据。就这么简单, 硬件 szyd。

但是我们发现, cache overhead 是一个很 under optimized 的东西:

我们之前的简单 Memory, 一共只有 16 格, 地址有 4 bits. overhead 比数据本身要小。

但是 what about 2^{32} , 2^{64} 个位置的 memory? 我们如果延续使用把整个地址作为 tag 的策略, 那么对于正常的有 2^{32} , 2^{64} 个位置的 memory 而言, 有 32 位/ 64 位的 memory address, 以 32 bit address 为例, 每一条 overhead 里就至少有 32 位的 tag.

再加上之后会有的 valid bit, dirty bit 以及 LRU bit 等等, overhead 会达到 40 bits 左右

而每一条的数据只有 1 byte = 8 bit. 因为这是 byte addressable 的.

这就导致了一条数据的 overhead 大小是数据本身的 5 倍. 于是整个 cache 就会非常臃肿。

优化 cache 就是优化它的 overhead. 我们需要想办法减小 overhead.

Reducing overhead

优化 overhead, 对于我们目前的 (fully associative) cache, 就是优化 tag 的大小.

我们目前的 tag 就是一个 address.

Idea:

对于一个完整的 address，把前面几位作为 tag，一个 tag 对应多个数据，形成一个 block.

比如我们在 **4-bit address** 中，把前三位作为 tag

于是 0000, 0001 都对应了 000 这个 tag. 一个 block 的 block size 变为 2 bytes: 即，每次我们都把两个 bytes 的数据放进 cache

Case 1:

Block size: 1 bytes

1	0	74
1	6	160

V tag data (block)

How many bits needed per tag?

$$= \log_2(\text{number of blocks in memory}) = \log_2(16)$$

$$= \underline{4 \text{ bits}}$$

$$\text{Overhead} = \underline{(4+1) / 8} = 62.5\%$$

Case 2:

Block size: 2 bytes

1	0	74	110
1	3	160	170

V tag data (block)

How many bits needed per tag?

$$= \log_2(\text{number of blocks in memory}) = \log_2(8)$$

$$= 3 \text{ bits}$$

$$\text{Overhead} = (3+1) / 16 = 25\%$$

我们自然得到计算一个数据的 tag 是多少的方法：

$$\text{tag}(x) = \lfloor \frac{\text{addr}(x)}{\text{block size}} \rfloor \quad (2)$$

比如数据在第 11 位，block size 是 2，于是 tag 是 5

对于一个 address，我们称它后面 tag 外的位数为 **block offset**.

比如 $11 = 0b1011$

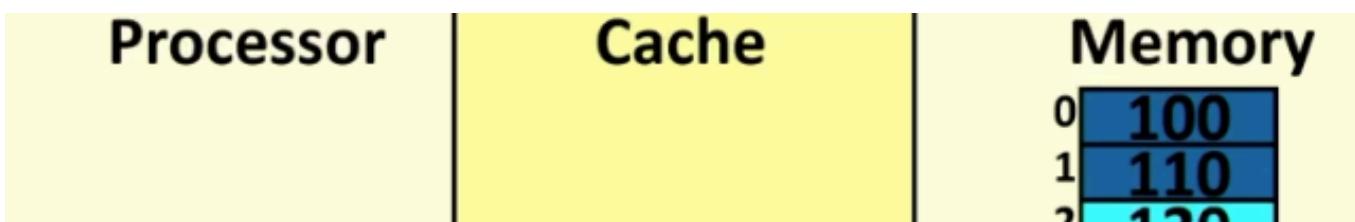
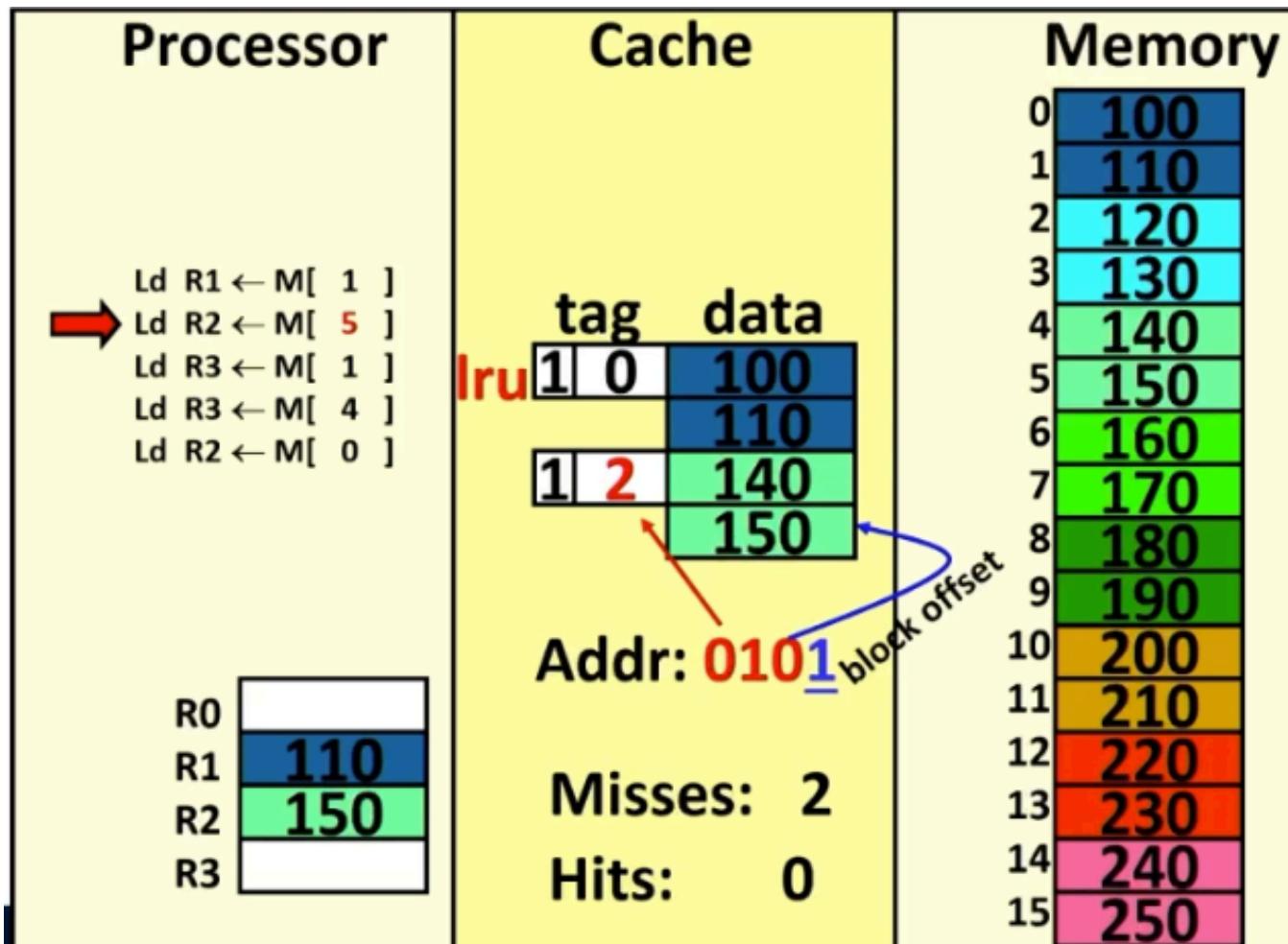
101 位 tag，最后一个 1 为 block offset

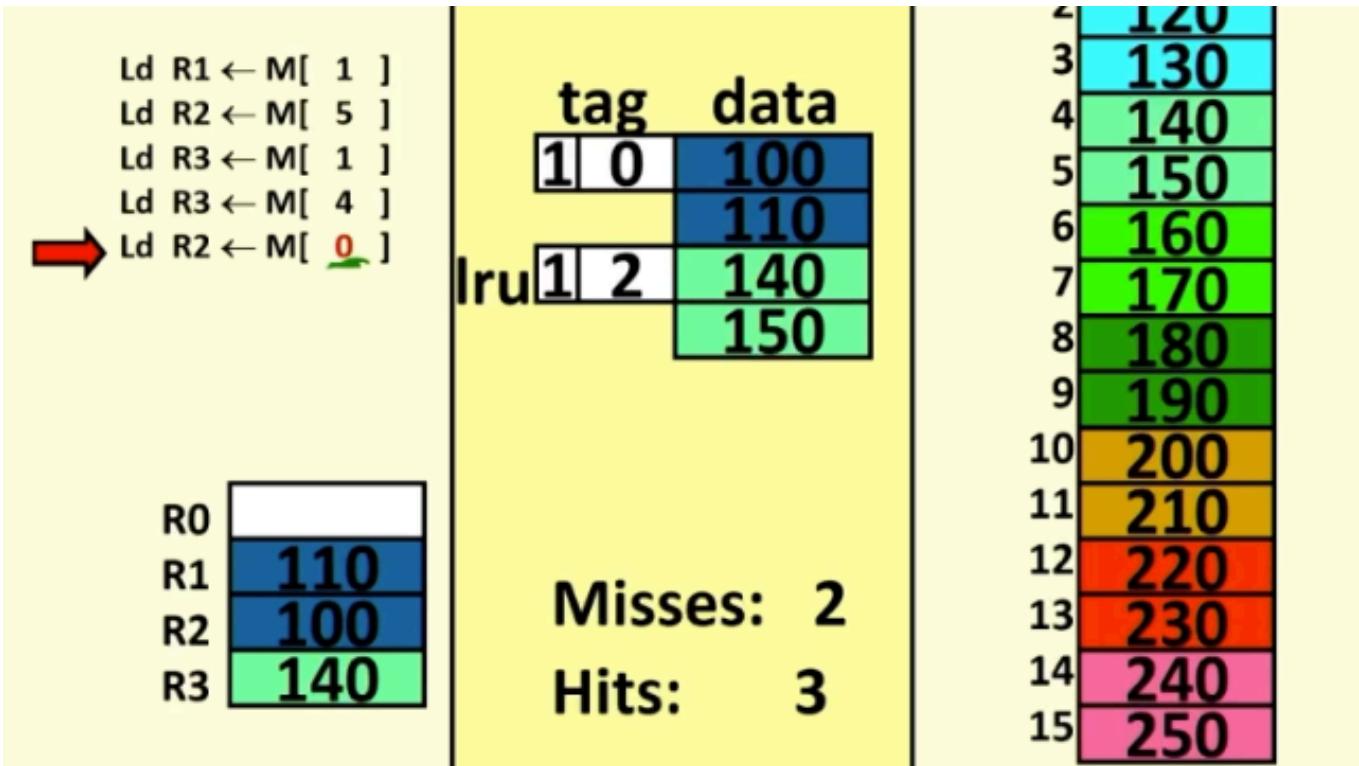
Spatial Locality

试想我们使用这个策略，用 3 bits 的 tag 来寻址 4-bit 的 address，每次都 grab 1 bit 宽度内的所有数据，也就是这里的两条数据 (2 bytes)

这个策略的意义其实比单纯的减小 cache 的 overhead 更多。因为它还很好地 利用了 spatial locality：因为我们每次 grab 的一个 block 中的数据都是连在一起的，通常程序中，连在一起的数据常常会先后紧接着被用到。比如数组的第一个元素和第二个元素。

这里举一个例子：这段 machine code



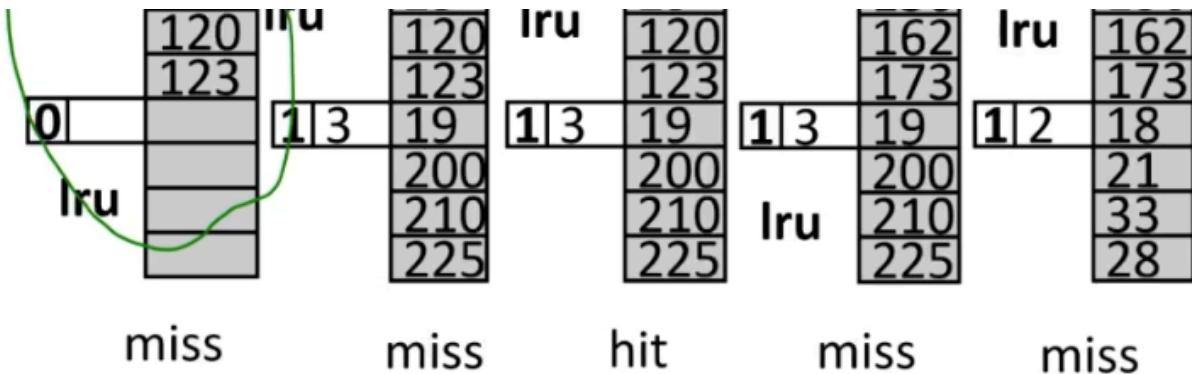


我们 load 了 m[1], m[5] 之后又 load 了 m[4], m[0]: 而 m[0], m[4] 已经在我们 grab m[1], m[5] 时连带着 grab 过了!

提高了 hit/miss rate.

Other examples: 如果使用 2 bit tag: 2 bit block offset, 4 bytes block

Ld R0 ← M[3]	Ld R2 ← M[12]	Ld R3 ← M[15]	Ld R1 ← M[4]	Ld R2 ← M[9]
V tag data	V tag data	V tag data	V tag data	V tag data
1 0 78 29	1 0 78 29	1 0 78 29	1 1 71 150	1 1 71 150
...		.		



LRU with more than two entries

目前为止我们的 cache 一共只有两个 entries，所以每次 access 其中一个，LRU 就是另外一个。

但实际的 cache 的 entries 数量远多于两个。所以我们需要给每个 entry 添加 LRU bits 来判断哪个 entry 是 LRU entry.

一共有多少个 entry, LRU bits 的表示范围就要是多少。有 N 个 entries, 那么 LRU bits 需要能够表示 0~N 的数字。于是就需要

$\lceil \log_2(N) \rceil$ 个 LRU bits

比如：如果 cache 一共有 4 个 entries, 就需要 $\log_2(4) = 2$ 个 LRU bits!

LRU bits 的值从 0 到 N-1, 0 表示这个entry 是 LRU 的, N-1 表示它是最 **most recently used** 的.

LRU bits 的更新算法如下：

- 一开始, entry 0, 1, 2, ..., N-1 的 LRU bits 分别赋予 0, 1, 2, ..., N-1 的值 (这样一来, 在 cache 第一次填满之前, 被 grab 的 blocks 是从前到后放进去的)
- 每当一个 element X 被使用: 给它赋予 N-1 的 LRU bits 值, 代表它是 most recently used 的
- 同时, 遍历所有 entries, 对于所有 **recent** 值比先前的 X 高的 entries, 把它们的 LRU bits 值 --, 这样就保持了整个 cache 里始终只有一个 entry 的 LRU 值是 N-1.

```
// when element i is used
X = counter[i] // first record the LRU value for comparison
counter[i] = N-1 // update LRU value of ele i as most recent
for (j=1; j<N; j++) {
    if(j!=i && counter[j]>X) counter[j]--;
}
```

}

ex:

Initial State				
Element	0	1	2	3
Count	0	1	2	3

Access Element 2				
Element	0	1	2	3
Count	0	1	3	2

Access Element 0				
Element	0	1	2	3
Count	3	0	2	1

Iteration is not actually costly

Note: 这里看似我们需要遍历所有 cache entries, 要花很多时间, 但是其实不用。

因为这是硬件。直接用 combinatorial logic, 用 mux 一步解决就可以。所有判断都是 Parallel 的
整个 cache 也是: 看起来我们要搜索一个 entry 要花费时间, 但是其实不花费时间, 都是并行的搜索。不像是写程序
是一个一个遍历要花 $O(N)$ time. 所以这里的这个类比程序其实不完全是对应的。

store: write back/through

我们之前都只考虑了 load word, 但是和 memory 的交互还有 save word 这个行为

当我们 store 一个 result 进入 memory 时, 由于 cache 的存在, 我们肯定要同步把 result 写进 cache 里。
但是我们还要考虑这一问题:

1. 如果它在 cache 中, 我们是否同步地把它写进 cache 和 memory 里 (write-through policy) ;
还是只写进 cache, 等到这个数据作为 LRU 被踢出来之后再放进 memory 里 (write-back policy) ?
2. 如果它不在 cache 里, 我们是否要把它放进 cache (allocate-on-write policy) ;

还是不这么做? (no allocate-on-write policy)

现在我们把 write word 也计入 hit/miss 里。allocate-on-write 即：在 cache 里找要 store 的数据的位置是否有，如果没有则是一个 miss，那么我们在 memory 里重新查询后，再把它放进 cache.

allocate-on-write 的性能优点是利用了 temporal locality：通常，刚刚 store 的 memory 不久之后也会用到。

而 write-back policy，看起来也比 write through 要更加 efficient 一点（未必，见下面分析），需要增加一个 bit 的 overhead：添加一个 dirty bit

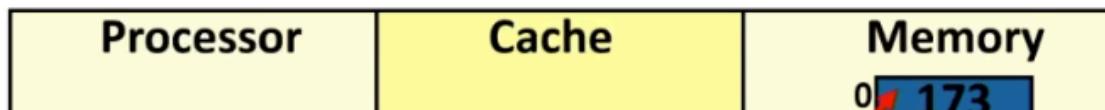
write-through policy

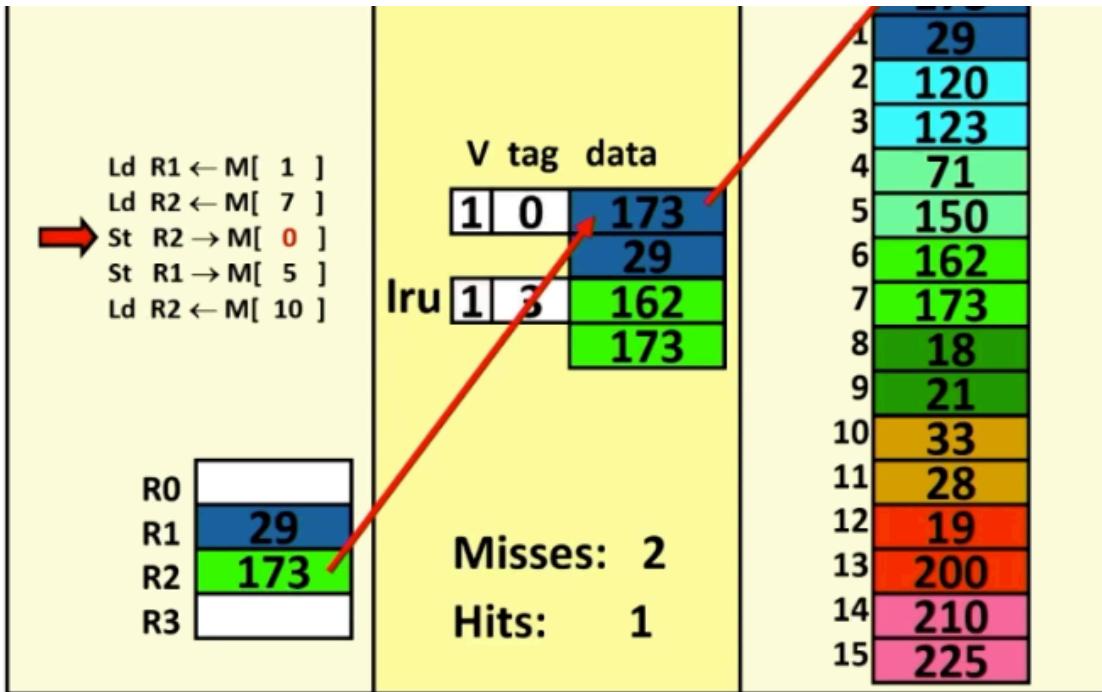
一个 write-through + allocate-on-write 的例子

1. load 了两个， miss
2. stote 找 $M[0]: 0 = 0b0000$, tag = 000, offset = 0, 在 cache 中找到了 tag 000，并 +0 获得了 data，把 R2 值传给它，并传给 memory.

Hit++

write-through, allocate on write (REF 3)

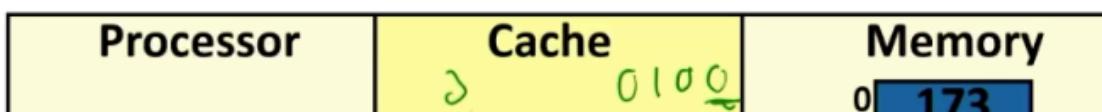


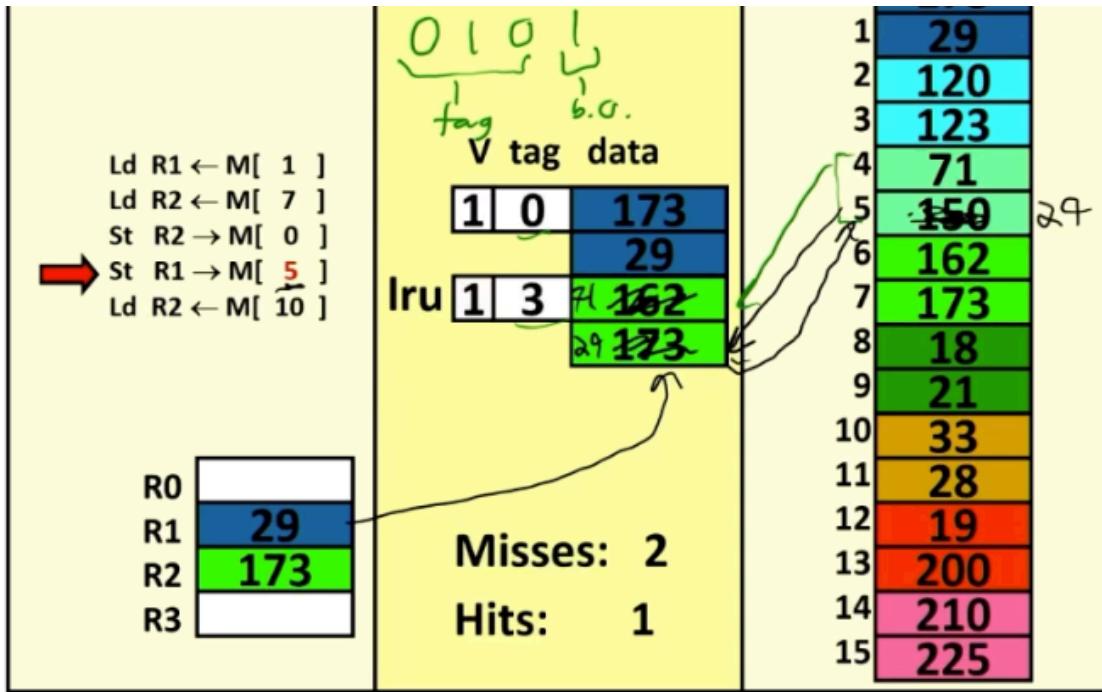


3. Store 找 $M[5]:5 = 0b0101$, tag = 010, offset = 1, 没有找到 tag, 于是前往 memory 找, 找到了之后首先改变 memory 中的值, 然后把整个改完之后的 $M[4], M[5]$ 作为 block 传给了 cache

Miss++

write-through, allocate on write (REF 4)





总计 memory references: 每次 miss 都往 memory 读两个 bytes; 每次 write 都往 memory 写一个 byte
 $2 \times 4 + 1 \times 2 = 10$ bytes

write-back policy

设置一个 dirty bit.

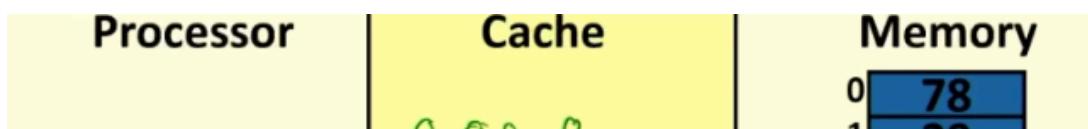
每次当一个 cache line 从 memory 被 allocate 进来的时候, 就 reset 为 0;

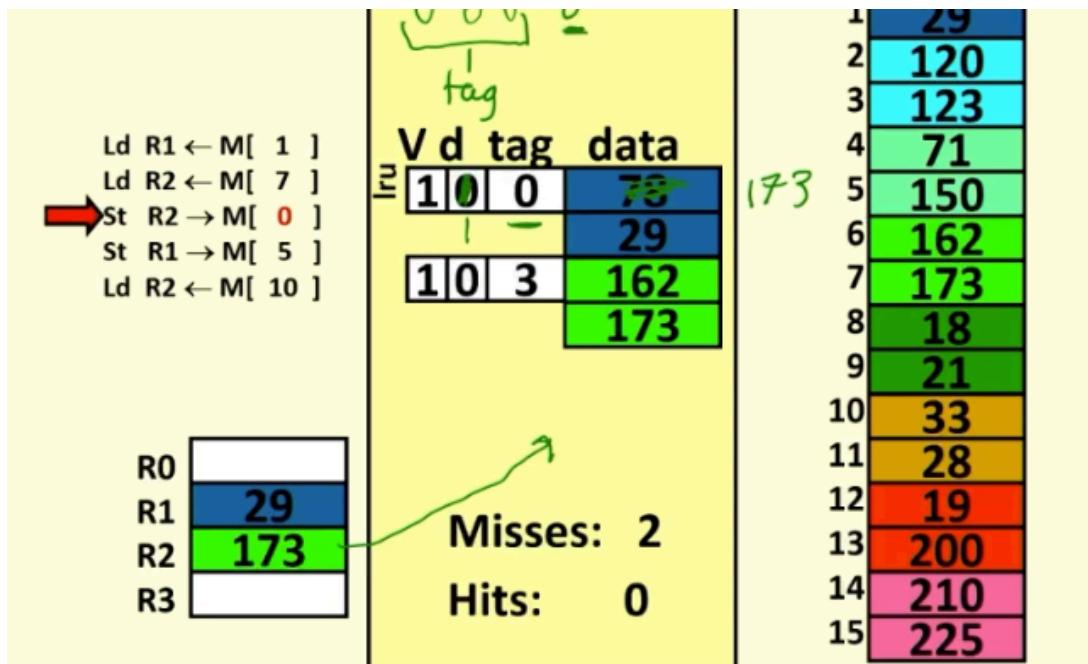
每当这个 cache line 被 reg store 进值的时候, 我们更新 dirty bit 为 1, 表示: 它被写入了数据, 但是数据并没有传回 memory, 所以这个 cache line 已经不是进来的时候的数据了, 是 "dirty" 的;

当一个 cache line 作为 LRU 被 evict 的时候, 我们检查它的 dirty bit: 如果 dirty bit = 0, 表示这个过程中我们没修改过它的值, 可以放心直接把它踢掉; 如果 dirty bit = 1, 表示我们修改过它的值了, 还要把它 send back 回 memory.

write-back + allocate-on-write 的例子:

当 write 数据进入 cache 时, 我们不 send 它 back to mem 而是更新 dirty bit. 在它被 evict 的时候 send 一整个 block back to mem.





性能对比

总计 memory references: 每次 miss 都往 memory 读两个 bytes; 每次 evict 一个 dirty line, 都往 memory 写一个 block: 这里是两个 bytes.

$$2 \times 4 + 2 \times 2 = 12 \text{ bytes}$$

对比刚才的 write-through policy 只交互了 10 bytes.

这个例子里我, write-back 的 memory reference 的总 bits 比 write-through 多, 所以 write-through 在这里反而更加 efficient! 这是因为 write-through 的每次交互都只是一个 bit, 而 write back 则是一整个 block, 因为 dirty bit 是一整个 block 的.

所以我们需要权衡考虑性能: write-back policy 适合当我们需要在短时间内对一个 particular address 多次写数据的场景。

Store w No Allocate	Write-Back	Write-Through
Hit?	Write Cache	Write to Cache + Memory
Miss?	Write to Memory	Write to Memory

Replace block?	If evicted block is dirty, write to Memory	Do Nothing
Store w Allocate	Write-Back	Write-Through
Hit?	Write Cache	Write to Cache + Memory
Miss?	Read from Memory to Cache, Allocate to LRU block Write to Cache	Read from Memory to Cache, Allocate to LRU block Write to Cache + Memory
Replace block?	If evicted block is dirty, write to Memory	Do Nothing

Lec 18 - Direct-mapped cache

我们目前为止搭建的 cache 都是 **fully-associative cache**: 任意 block in memory 都可以 go to cache 的任何位置。因为我们的实现方式是把 address 完整地分为 tag 和 offset. 这使得我们的寻址行为可以在整个 memory 中进行。

代价是: 在比较大的地址空间里, 要么 overhead 过长, 要么 block 太大导致 copy 的成本太大。

比如 32 位地址空间, 我们不得不把地址二分为 tag 和 offset

如果我们使用 22 位的 tag, 那么我们的 offset 大小就是 10 位, 也就是说: 一个 block 里面包含了 $2^{10} = 1,024$ 个 bytes 的 data, 把这些 data copy 过来的代价太大了

map a mem block to a cache line

现在考虑一种新的策略:

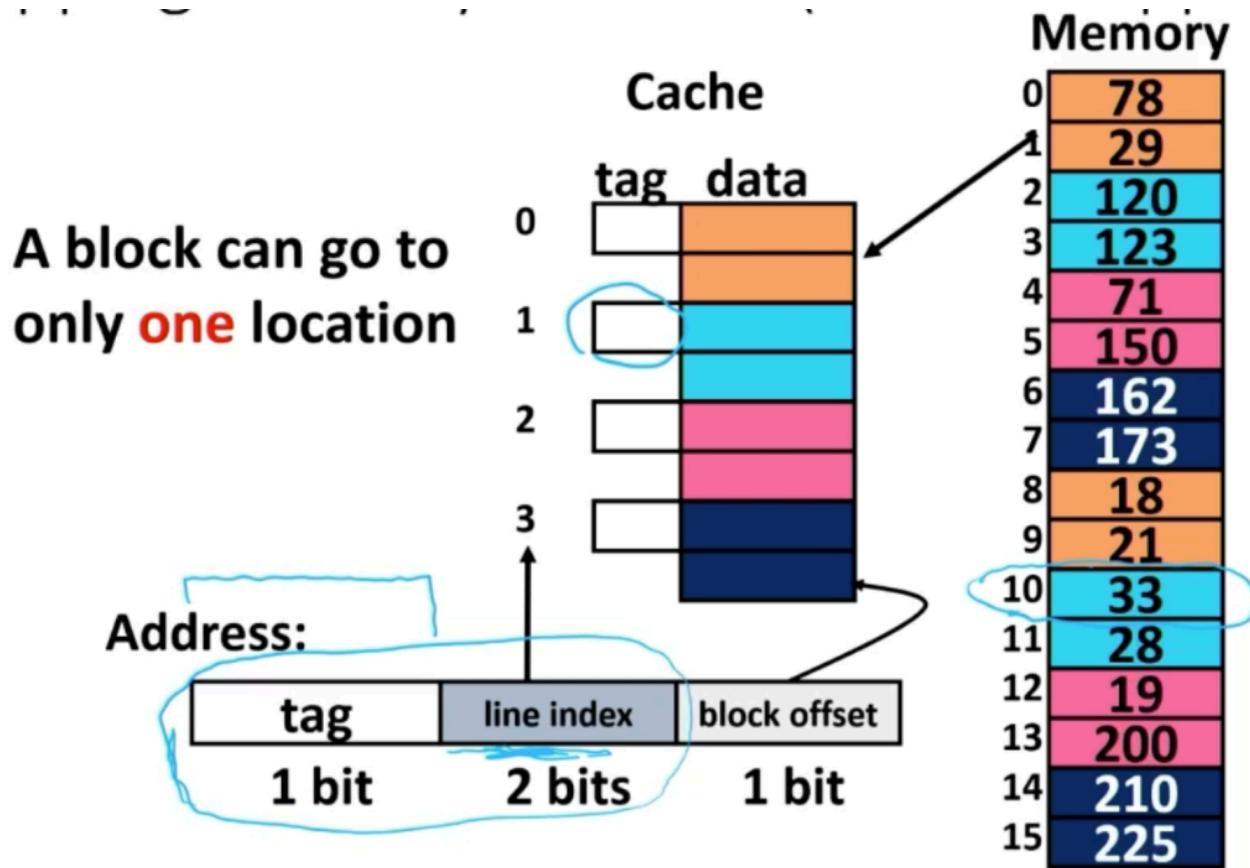
direct-mapped cache, 每个 block in memory 只能去 cache 里面固定的一行.

也就是说, 这个 map 是一个从 memory 到 cache 的函数. 一个 mem block 被 map 到一个 cache line, 属于这个 line 的 addressable space.

Idea:

1. 对于 N 行的 cache, 我们把 memory 跳着划分成 N 份:
2. cache 的 lines 为 0, 1, ..., N-1;

3. 对于 memory, 上一个 block 划分进 line n 的 addressable space, 下一个就划分进 line $n + 1 \bmod N$ 的 addressable space.



划分一个 address

而具体的做法是：我们把一个 address 分为 tag, line index 和 block offset 三个部分

line index 不需要称为 overhead, 只要在 cache grab from memory 的时候 grab 特定的 memory blocks 就可以了
line bits 的数量就是 $\log_2(\lceil \text{cache lines} \rceil)$

ex:

$m[12]$, $12 = 0b1100$, 其 tag 为 1, line index 为 $0b10 = 2$, block offset 为 0

所以 12 属于 cache 的 line 2 的 addressable space, 只能 go to cache line 2; 而它在 cache line 2 中的 tag 为 1, 因为 tag 0 的 block 是 $m[4], m[5]$; 而它在自己的 block 中的位置是 0.

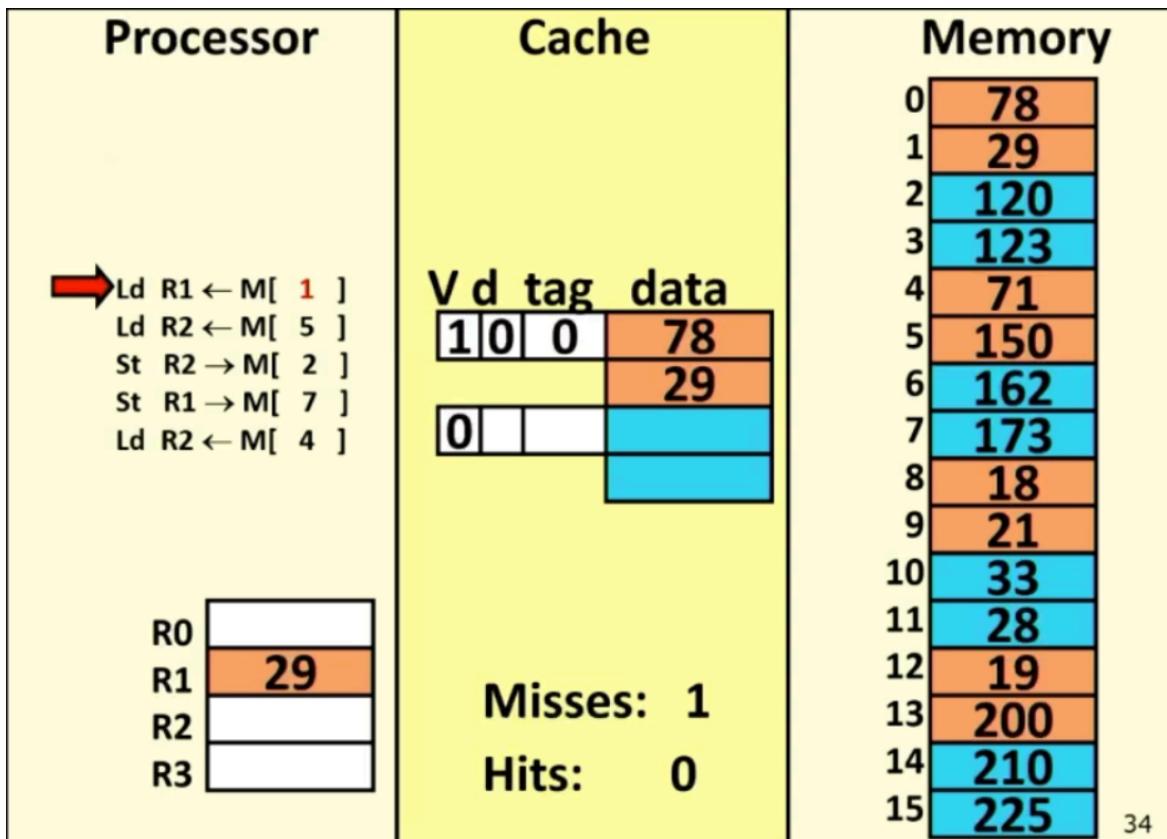
example:

cache 有两行, 于是 line bits = 1.

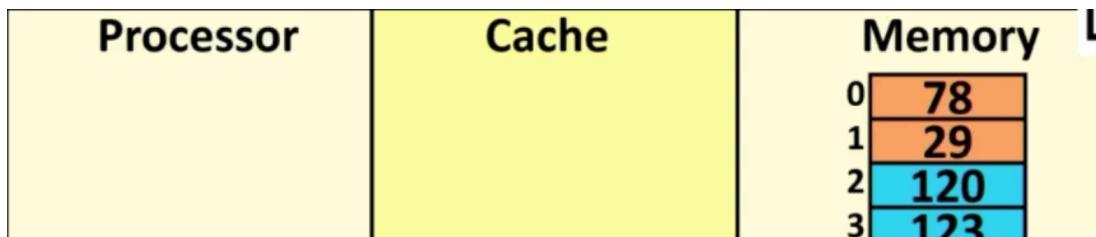
所以还剩下 3 bits 分给 tag & offset

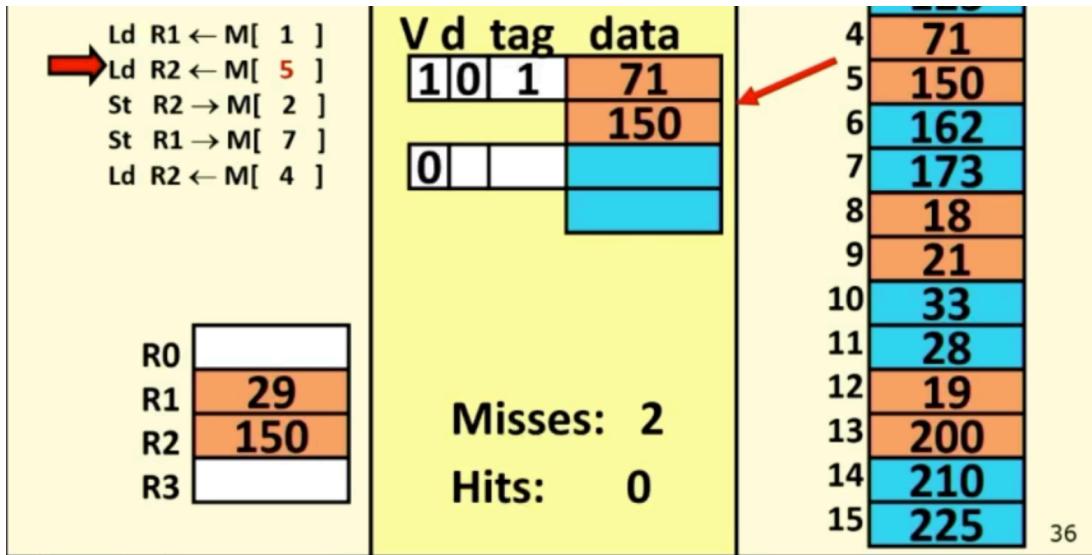
既然 block size 为 2: offset bit = $\log_2(2) = 1$

所以 tag bits = 3-1 = 2, 每个 cache line 对应了 memory 中的四个 block.



1. load from M[1], 1 = 0b0001, line 为 0, tag = 0b00 = 0, 查看 line 0 并没有发现这个 tag, 于是 grab from M[0:1], 放进 cache[0] 里.





2. Load from $M[5]$, $5 = 0b0101$, line 为 0, tag 为 $0b01 = 1$, 查看 cache[0] 发现 tag 为 0 而不是 1, 于是 grab from $M[4:5]$, 放进 cache[0] 里.

Note: direct-mapped cache 不需要 LRU bits

因为一个数据只对应一个 cache line! 读到需要查看的 address 时, 到它对应的 cache line 查看, tag 对就 hit, tag 不对就立刻替换.

direct-mapped cache 的优势是对于大的 address space 处理得更好, 但是缺点是比起 fully-associative cache 显然更容易发生 conflicts.

不过, direct-mapped cache 很好地利用了 spatial locality. 虽然看起来它很容易 miss, 但是同一块连续的 memory blocks 可以完全不冲突地放入 direct-mapped cache 中。这是因为连续的 blocks mapped to 的 cache line 不同, 正好能够被整个 cache 容纳

stack frame 的 growth 就是连续的, 所以这很符合实际的程序设计需求。

Lec 19 - Set-Associative cache

Fully associative: any block can go to any cache line

Direct mapped: divide memory into regions, **map a region to a cache line**

set associative: divide memory into (fewer) regions, **map a region to a set of cache lines**

set associative 是介于 fully associative 和 directed mapped 之间的 cache design, 或者说 :

fully associative, directed mapped 都是特殊的 set associative cache.

1. **fully associative** 是整个 cache 都是同一个 set, 整个 memory 都是同一个 region 的 set-associative cache
2. **directed mapped** 是每个 cache line 都是一个 set (one way), 整个 memory alternating region 的 cache

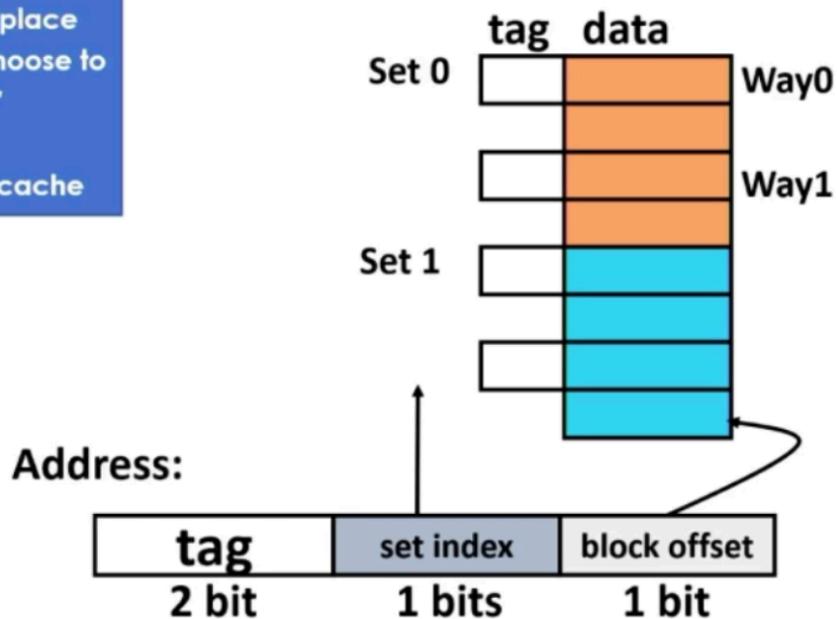
fully associative cache gives **best hit rate**, 但是对于大的 address space 而言, 需要的 entries 过于多, 以至于速度慢下来。

Direct-mapped cache 更能处理大的 address space, 但是 hit rate 降低。

而我们可以选择合适的 set 大小, 建立 set associative cache, 效果介于 fully associative 和 Direct-mapped 之间

Set-associative cache

"Way" means "place hardware can choose to put data"
This is a 2-way cache



Memory	
0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

一个 n-way set associative cache 的意思是: 同一个 set 里有 n 个 entries.

一个 set 就相当于一个 fully-associative cache.

计算一个 n-way 的 fully-associative cache 有几个 set: num of set = $\frac{\text{num of entries}}{n}$

How set associative cache divide address space

一个有 m 个 sets 的 set associative cache 把 address space 分成 m 份

和 direct-mapped cache 一样，我们把一个 address 分成三个部分：

1. Tag bits
2. set bits (line bits in direct-mapped, 因为 direct-mapped 中一个 line 就是一个 set!)
3. block offset

$$\text{block offset bits} = \log_2(\text{blocksize})$$

$$\text{set bits} = \log_2(\text{sets})$$

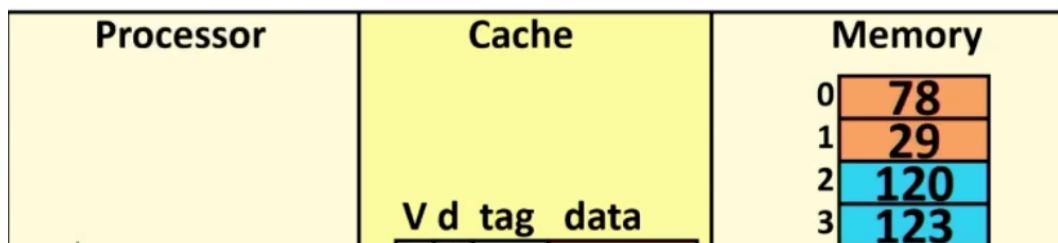
tag bits: the rest

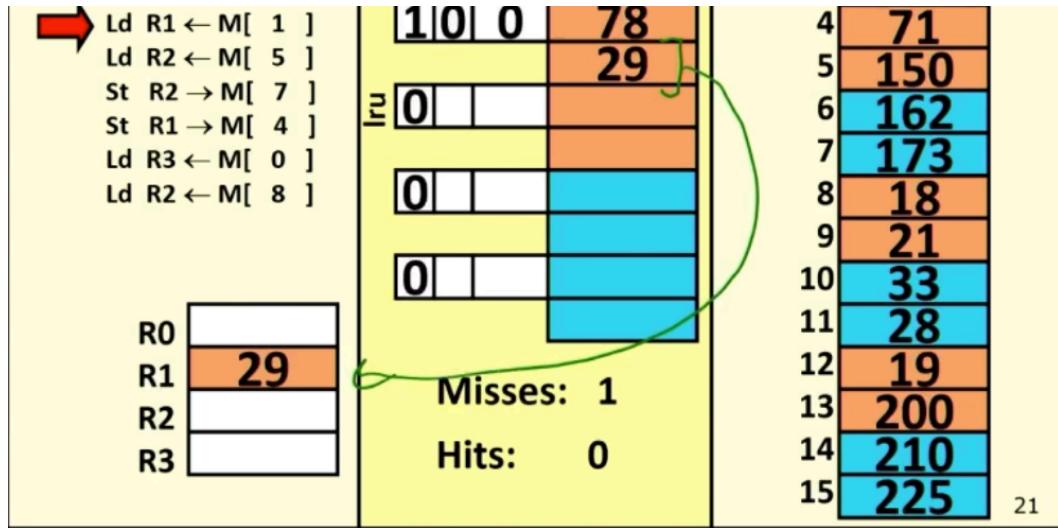
比如这里：我们一共有两个 set，于是有 $\log_2(2) = 1$ 个 set bit

$$\text{block offset bits} = \log_2(2) = 1$$

所以有 2 个 tag bits

example

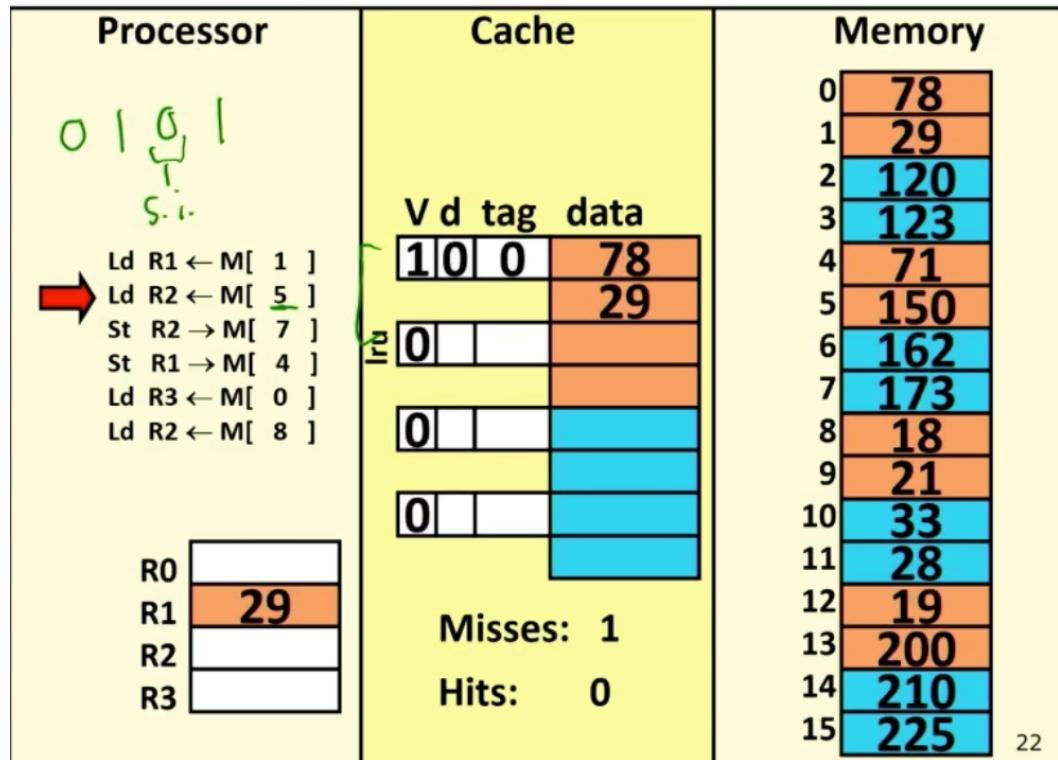




1. load M[1], 1 = 0b0001, set 0, tag 0, offset 1

于是前往 set 0 寻找，并没有发现 tag 0

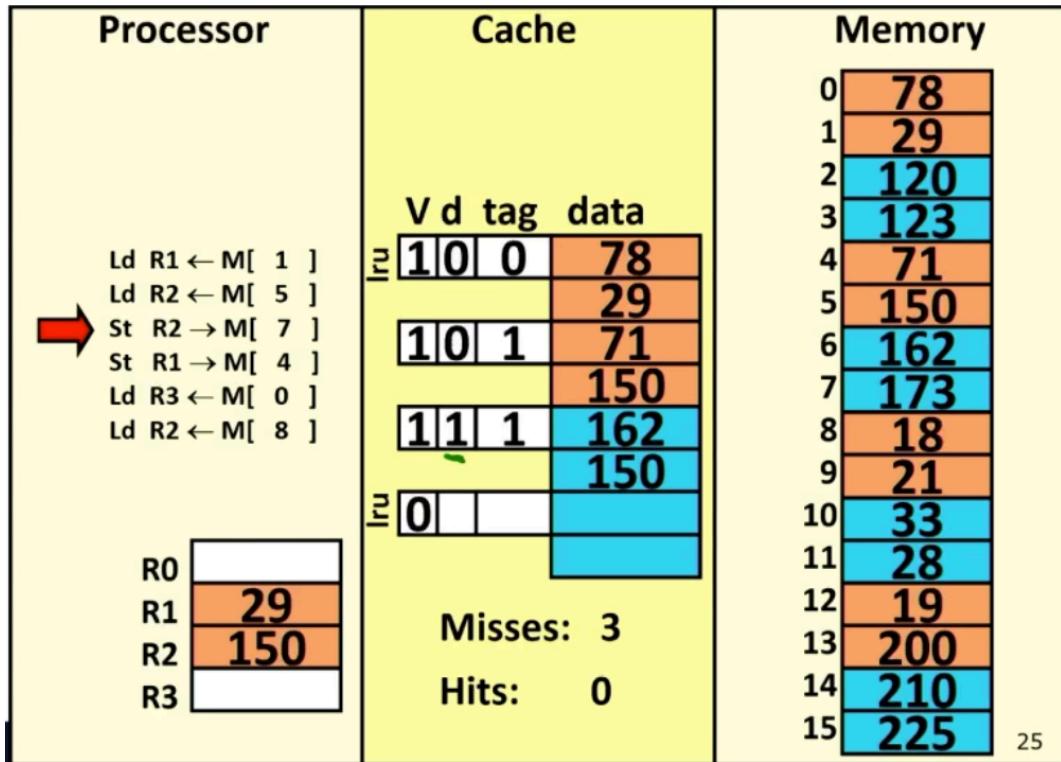
于是前往 memory 中 grab set 0 的 region 中的第 0 个 block.



2. Load M[5], 5 = 0b0101, set 0, tag 1, offset 1

于是前往 set 0 寻找没有发现 tag 1

于是前往 memory 中 grab set 0 的 region 中的第 1 个 block.



25

3. Store 到 M[7], 7 = 0b0111, set 1, tag 1, offset 1

前往 set 1 没发现 tag 1, 于是前往 memory 中 grab set 1 的 region 中的第 1 个 block, 放进 cache

并且注意: 这是个 store! 于是在 cache 里修改这个 block 的 offset 1 的元素, 并设置 dirty bit 为 1, 以标记把它 evict 的时候要修改 memory.

Practice problem

Byte addressable 的空间

32 bit address (note: 我总是搞混的概念, , , 32bit 指的是一共有 2^{32} 次方个 address. 每个 address 都是一个 Byte, 0000 0000 0000 0000 0000 0000 0001 代表位置 1 上的这个 Byte, 0000 0000 0000 0000 0000 0000 0000 0000 0011 表示位置 2 上的这个 Byte, 不是一个 address 上的东西的大小.)

$64 = 2^6$ B blocksize

$16KB = 2^{(4+10)} = 2^{14}$ B cache

因而 cache 里一共有 $2^{14-6} = 2^8$ blocks, 每个 block 有 16 行

blockoffset: $\log_2 64 = 6$ bits

在所有的 cache 设计里, blockoffset 都是一样的, 只取决于 block 大小. 表示一个当前的 address 是它所在 block 的第几个地址, 因而 blockoffset 的 bit 数即用多少个 bit 的二进制数可以表示一个 Block 中能容纳的地址数量

32 位 address 分为 tag, set index, offset, 即 tag

The breakdown of the address:

1. fully associative:

set index: 0 bit

tag: 26 bit

2. 4-way set asso cache:

set index: 2 bit

4-way: 一个 Set 里面有 $4 = 2^2$ 个 block! 因而一共有 2^{8-2} 个 sets, 也就是 2^6 个, 需要用 6 bit 表示

因而 set: 6 bits <https://drive.google.com/drive/my-drive>

因而 tag: 20 bits

3. direct-mapped cache:

一个 block 就是一个 Set.

一共有 2^8 个 Sets, 也就是说 8 bits 的 set index

tag 有 $32-6-8=18$ bits

More Sophisticated: Ln Cache, I/DCache

cache 变大的代价就是变慢

我们希望 cache 在足够大和足够快之间有一个平衡。

因而实践中我们设计多个 caches, 让紧急的 data 放在最近的 L1 cache, 使用比较少的 data 放在 L2, L3 cache

要把 cache 的设计放进我们的 Pipeline processor, 我们需要把 Instruction 也拿出一个 cache 来放。

否则, instructions 是从 memory 里取的, 而 data 则由于有 lw, sw 可以和 cache 有交互, 因而 cycle 时间不平衡

对应的方案是: 我们分 ICache 放 instructions, DCache 放 Data

(具体: 如果其中一个 Miss 了怎么办? 如果两个都 miss 了怎么办? 我们在这节课先不考虑)

Lec 20 - Classifying Cache Miss

我们应该如何设计 cache 的大小和 associativity 等属性，从而改进一个 cache？要改进一个 cache 就是改进它的 miss rate。

因而我们首先要对 miss 进行分类

3 reasons for cache misses

1. compulsory miss: never accessed this data before
2. capacity miss: 由于 cache 不够大导致的
3. conflict miss: 由于 restrictive associativity 导致的；cache 足够大但是一个 set 太小，导致了 conflict

capacity miss 和 conflict miss 之间的界限比较模糊。

我们这样定义：

1. 如果 cache 比 memory 还大（或者说无限大）的情况下一个 miss 仍然会发生，就是 compulsory
2. 如果改用 fully associative cache 就不会发生这个 Miss，就是 conflict；
3. 否则就是 capacity

Practice problem

假设我们有 64B cache, 16B blocks, 2way, 2sets

(两个 blocks 在 set1, 两个 blocks 在 Set2; 每个 block 16 行)

note: 由于一个 block 16 行，我们可以用 hex 数来表示 address，这样的话第2位 hex 不一样就说明不在一个 block 里)

由于有两个 set, hex 地址第二位是 even number 则在 Set 0, 第二位是 odd number 则在 Set 1

Address	Infinite	FA
0x00	M	M CRU 3012 0
0x14	M	M 2301 01 1230

0x27	M	M	012
0x08	H	H	012
0x38	M	M	0123
0x4A	M	M	0423
0x18	H	M	0231 0413
0x27	H	M	3120 2413
0x0F	H	M	2013 2410
0x40	H	H	1302 2410

Address	Infinite	FA	set ₁ SA set ₂	3Cs
0x00	M	M	10 M 01	Compul
0x14	M	M	10 M 10	Compul
0x27	M	M	82 M 10	Compul
0x08	H	H	10 02 H 10	—
0x38	M	M	02 M 013	Compul
0x4A	M	M	04 M 013	Compul
0x18	H	M	01 04 H 10	—
0x27	H	M	10 24 M 103	Capacity
0x0F	H	M	01 20 M 103	Capacity
0x40	H	H	10 40 M 103	conflict

(note: set asso 的 hit, full asso 也可能 miss!)

减少 Miss 的 3 个对应方法

减少 compulsory miss: 增加 block size, 单次 grab 的时候连带 grab 更多 blocks

缺点: given cache size, 增加 Block size 会减少 block numbers 从而减少灵活度

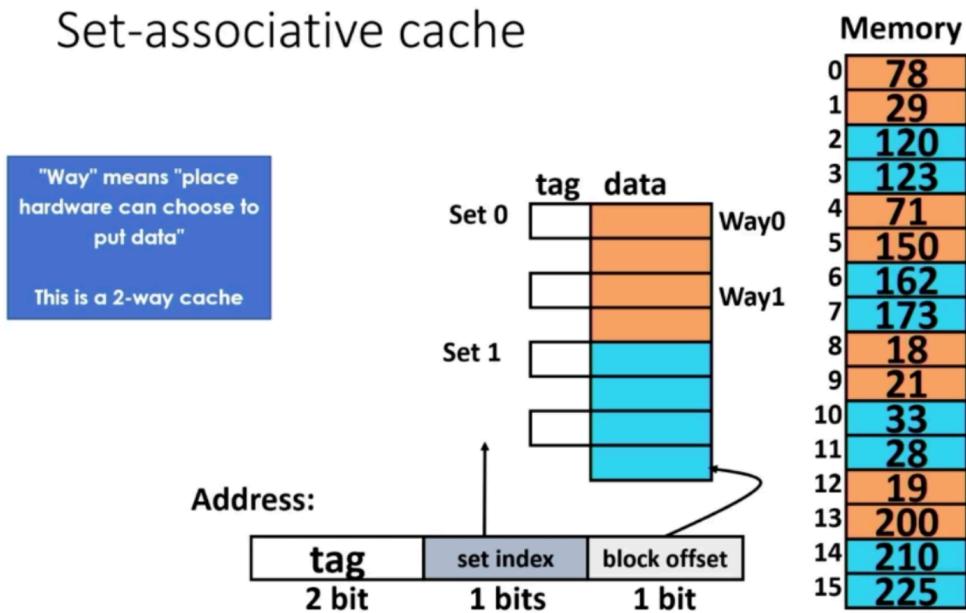
减少 capacity miss: bigger cache

缺点: 会增加 latency

减少 conflict miss: 更大的 associativity

缺点: 会增加 latency 以及 Overhead (因为 ways 数量每翻倍, set 的数量都减半, block offset 不变, set bits 减少一个, tag 的 bits 数就增加一个)

tag bits 的增加就增加了 overhead, 并且检索量翻了一倍, 增加了 latency

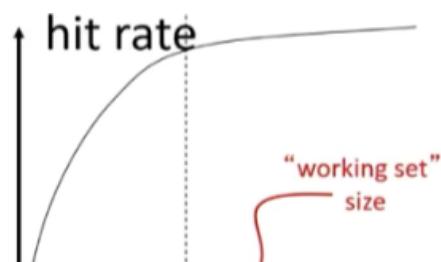


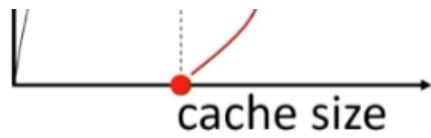
Cache size 的影响

Cache size 的增加会增加 latency, 也会增加 Hit rate

太小的 cache size 的 hit rate 太低, 会经常要更改 cache line, 导致利用不好 temporal locality

所以会呈边际递减。cache size 应该有一定大, 但是不能过大, 否则 miss latency 的增加就会超过 Hit rate 的增加带来的时间消耗的减少



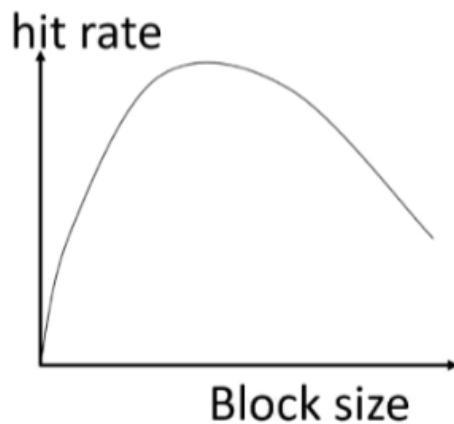


Block size 的影响

block size 代表对 spatial locality 的假定程度

太小的 block 会利用不好 spatial locality, 并且也会增加 tag overhead (block offset 减小, tag bits 增加)

太大的 block, 在给定 cache 大小下, 会减少 blocks 的总数量; 首先, 大 block 会增加 data transfer 一次的数量; 其次, blocks 总数量的减少, 对于多个比较远的 memory 位置上的频繁交替调用 (比如函数调用), 会导致频繁的 data transfer, 增加了 Miss rate

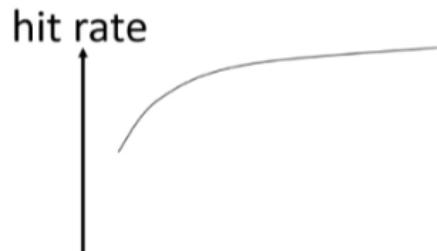


Associativity 的影响:

associativity: 多少个 Blocks 可以 map 到同一个 set?

更大的 associativity 会减少 miss rate, 会降低在不同 programs 中切换的 variation

更小的 associativity, 虽然 miss rate 会增加, 但是 hit time 会减少: 每次只需要搜寻更小的 space, 只在一个set 中就可以了





Practice problems

Pipelining with cache:

remember that :

$$CPI = 1 + \text{stall possibility} * \text{stall cycles}$$

一个 Clock cycle 如果 500Mhz: 2ns cycle time

如果 cache miss 有 100 ns latency: 等于 50 cycles

判断 block size, associativity 以及集合数:

block size 是最先看出的。并且一定是在最前面几行看出，因为后面的 Miss 可能有 associativity 的缘故，前面一定因为 Block size

我们通过相近的 miss 来判断 block size 的上界，相近的 hit 来判断下界

remember: 必须是 2 的倍数

associativity 其次看出。有一定难度

N-way: 一个 set 里面有 N 个 Blocks, 容量是 N

direct map 首先要经历排除: direct map就是1 way, 那么 cache 中一个 block 对应一个 set。

因而, 一个 address 过了 cache size 个 addresses 之后遇到和它相同 set 的 index

比如 cache size 512B, 那么 0x310在过了 512B, 也就是 $2 * 16^2$, 到了 0x510 之后才是同一个 index

如果同一个 index 上的元素连续 hit, 那么就说明 #way > 1!

其次, 我们可以 Bound 住 #ways. 通过观察, 同一个 Block 的元素, 在 Hit 之后经历了多少个其他操作被踢了出来。它被放进去的时候是 LRU 值最高的, 如果经历了两个指令就被提出来了 (同 block 的元素 miss 了) 那么一定在这两个指令之间被替换了, 所以 block size ≤ 2 .

目前我们的讨论：假设所有 Programs 都对 memory 有 full, private access

但是实际上：多个 programs 同时运行。试想：两个 Program 可能同时都想 write to 同一个 memory address

并且，即便我们只有一个 Program 在运行。我们知道，既然我们的地址有 64 位，寻址空间有 2^{64} 个 bytes，我们可以搜寻其中任何一个 Byte 的地址。也就是说，一共有 2^{24} TB 的储存单位在 memory 里？我们理应可以输入 2^{64} 个地址中的任何一个

这显然是不可能的。实际上 modern system 对这两个问题给出了同一个答案，也就是 virtual memory：

每个 program 都假定它有 full private access to memory，也就是能够寻址 0x0 to 0xFFFF...FFFF

而我们的 hardware 和操作系统的 software 把这些虚拟地址 map 到 DRAM 以及 hard disk 上的具体的不同位置

所谓 virtual 就是使用 a level of indirection

Address translation 的工作在 software 端由一系列合称为 operating system 的 programs 来管理，它们直接管理 **hardware resources for all other running programs!**

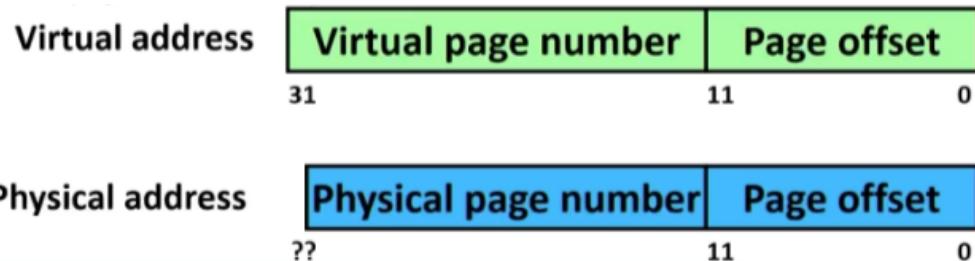
operating system 实现 address translation 是依靠一种叫做 **page table** 的 data structure.

Pages

Memory 被划分为 fixed-size chunks 称为 pages (4KB for x86)

一个 virtual page 的大小和 physical page 的大小相同

一个 virtual address 被分为 page number 和 page offset 两部分，就像是 cache 中一个地址被分为 tag 和 block offset 一样



这样之后，翻译一个 virtual address into physical address 实际上就是把 virtual page number 映射到 physical page number (like hash, modular operation, 由于 Physical page number 有限, virtual page number 比较多, 这不是一个 injective map

pages 的 boundary 上，相邻的 virtual address 可能会被 map 到 physical 距离很远的不同 pages 里，但是在同一个

page 里, contiguous 的 address 仍然保持 contiguous

Question: why pages? 为什么不直接 map 一个 virtual address to 一个 physical address 呢?

Answer: This is actually just talking about page 的大小. 这个 Idea 等于 Have 1B page.

pages 越小, 那么数量范围就越大。

比如 2^{64} B 的 virtual memory, 一个 page $4KB = 2^{12}$ B, 如果是 single level page table (之后我们会知道不可行, 要使用 multi level 才实际), 那么 pages 的范围是 0- 2^{52} , 所以 page table 有 2^{52} 这么大! memory 里根本放不下。如果一个 page 1B, 那么 page table 的大小是 2^{64} , 更加不可能

并且, 我们的 Page 大小也要足够, 以运用 spatial locality of a program. 否则跨 page 找 data 的 cost 会很大

Page Table

刚才说到, 翻译一个 virtual address into physical address 实际上就是把 virtual page number 映射到 physical page number, which is not injective; 具体实现这种映射的方法是 page table

一个 page table 就是 virtual page # 到 physical page # 的映射规则

每个 running process 都有自己的 own page table, maintained by OS

page table 自身在 memory 中, OS 知道 page tables 的位置 (所以我们假设 OS 运行正常, 不用管怎么 access page table)

Page table 就长这样:

第 n 行表示第 n 个 virtual page, 这行的 entry 是一个 valid bit 加上 n 对应的 physical page num

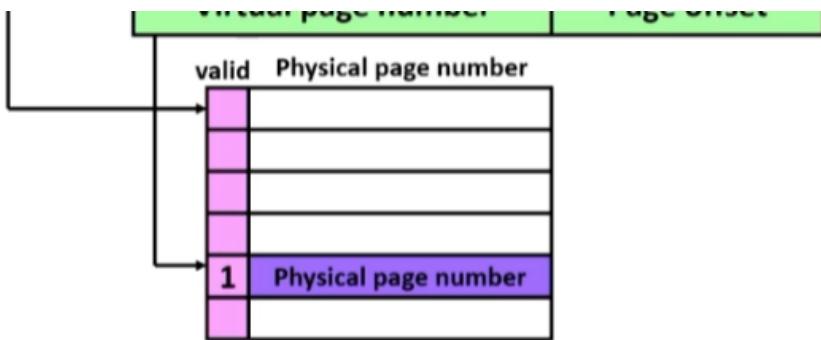
我们有一个专门的 **page table reg** 指向 Page tables 的 beginning; 可想而知, 我们执行程序的时候, 一个 process 有一个自己的 page table, 因而 page table reg 会不停地切换多个 Process 的 page table 的 beginning!

(recall: 同一个 processor 的同一个 kernel 可以管理多个 process, 这是通过 OS 把时间分为小的 slices, 时间片用完后, 操作系统会暂停当前进程, 并切换到下一个进程, 这种快速切换 (通常以毫秒级或微秒级) 给人的感觉是多个进程在“同时”运行)

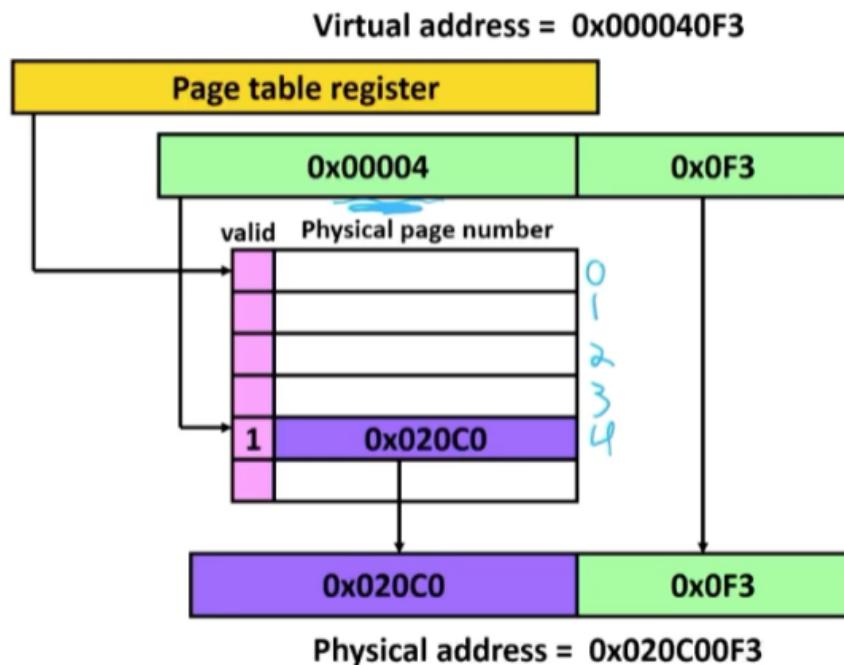
Page table components



The **Page table register** points to the beginning of the page table in main memory



ex:



4 KB page: 2^{12} B, 对应 12 bits 大小表示, 也就是 3 hex

于是一个地址的后三个 hex 位是 offset, 前面是 page num

Extend VM to disk

Virtual memory 的目标:

1. Transparency: 一个 Program 不用管其他 programs, 可以调用整个 virtual memory;
2. protection: 一个 program 不能 access 其他 programs 的 data
3. capacity: 每个 program 都可以拥有比 DRAM size 更多的 data, 意味着我们可以把 physical memory 扩充到 hard disk!

Transparency, protection 的实现就是保持 Physical pages 不冲突，两个 program 不能有 map 到同一个 physical table 的 virtual table 就可以

capacity 的实现就是：当 DRAM 用完的时候用 disk 作为 temporary space

可想而知这很慢。

所以这就解释了我们的电脑突然卡顿一下的原因：

如果我们的电脑持续稳定地卡，说明cpu不行；但是如果经常突然卡一会儿，说明 memory 太小，因为卡一会儿的原因是 processes 太多，共计 pages 已经占满了整个 memory，于是 extend VM 的 physical 映射目标 to disk，用 disk 来存新的 table，由于 disk 比内存慢几百倍，这个时候存取数据就会非常慢；关掉一点程序就好了

valid bit 的使用

并且：notice，如果真的在 disk 和 reg 之间交互会非常慢，我们尽量只交互一次。即：每当要处理 hard disk 上的 page，我们把一个 memory 里的 page 和 hard disk 里的 page 调换。

我们通过这种方式来告诉 hardware，这个 page 不在 memory 里而是在 disk 上：我们把 valid bit 设置为 0，并且后面跟上它 memory 里的位置

valid bit 为 0 有两个情况：情况 1，这个 **virtual page** 并不对应任何 **physical page**，还没有被 **assign**。这个情况，**valid bit = 0** 并且后面跟着的 **page num** 也是 0，是空的

情况 2，这个 **virtual page** 对应的 **physical page** 在 **disk** 上。这个情况，**valid bit = 0** 并且后面跟着的 **page num** 不是 0，是具体的 **disk** 位置

检测到 physical page 在 disk 上后，我们执行这个流程：

1. **Stop this process:** The current process is paused because it cannot proceed without the requested page.
2. **Pick page to replace:** The system selects a page in physical memory to be replaced (using a page replacement algorithm, e.g., LRU or FIFO).
3. **Write back data:** If the selected page is dirty (modified), its contents are written back to the disk.
4. **Get referenced page:** The required page is loaded from disk into physical memory.
5. **Update page table:** The page table entry is updated to reflect the new physical location of the page, and the valid bit is set to 1.
6. **Reschedule process:** The process that caused the page fault is resumed.

通过这个流程，替换 disk 和 memory 上的一个 page。

和 cache 里，sw 的 write back 很像

Lec 22 - Multi-Level VM

Size of page table

physical page numbers 的数量取决于 **physical memory** 的实际大小

假设我们有 $2^{30}B$ 的 Physical memory, $4KB = 2^{12}B$ 的 page size

那么 physical page num 的范围是 $0 \sim 2^{30-12}-1$, 需要 18 bits

page table 一个 entry 占 bits 数是 **physical page num** 的 bits 表示 + 其他功能性 bits, 比如 valid, dirty, read only 等, 算它 6 bits, 因而一个 entry 大小是 $18+6 = 24$ bits = 3B

而 **page table** 的 entry 数量就是 **virtual pages num**

假设是 32 位系统, 那么就是 $2^{32-12} = 2^{20}$ 个

page table size = entry 大小 * entry 数量 = $2^{20} * 3B$ 约等于 3MB

看起来不算太大, 但是如果我们改成 48, 64 bit 的 system, 就变成了 TB 级别。

Multi-Level Page Table

我们想要减少 Page Table 的大小

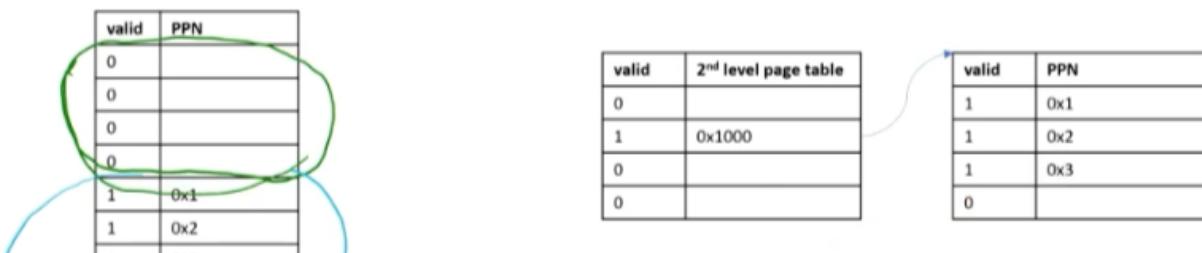
Idea: 在 16 位的系统里, 一个程序通常会使用大部分的 virtual memory space.

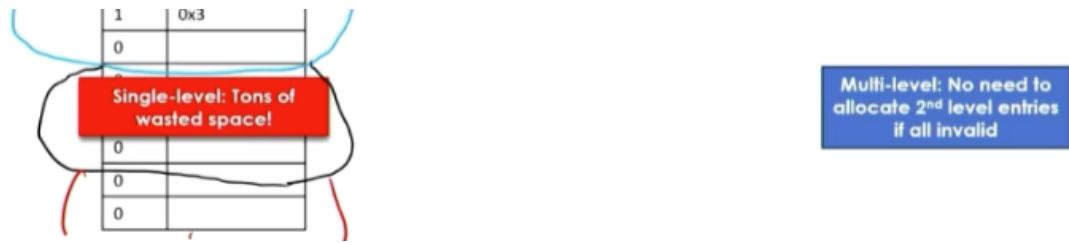
在 32 位的系统里, 一个程序会使用一部分 virtual memory space

但是在 48, 64 位的系统里, 一个程序通常只有使用一小部分 virtual memory space, 在非常极端的情况下才会使用大部分的 virtual memory space.

没有用到的 table entry: 全部都空着

所以我们把整个 **page table** 也 slice! 每次当当前的 **Page table** 全部爆满的时候, 我们才新建一个 **chunk of table entries**.



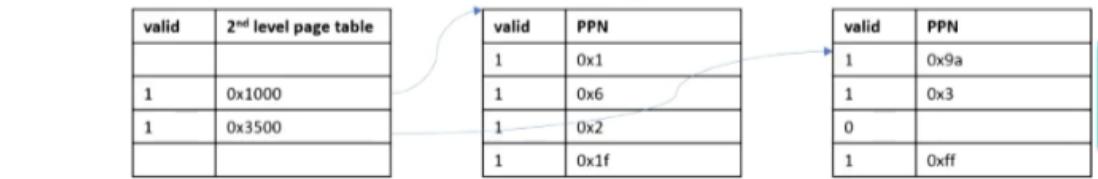


Data Structure

实现：我们用一个 1st level 的 page table 来储存 addresses of 2nd level page tables；我们只 allocate space for 2nd level page tables in use.

也就是说，当当前分配的 page tables 都满了，我们就在 first level page table 上新建一个 entry，并给它分配一个新的 second level page table.

As we access more, second level page tables are allocated



我们只有要使用一个二级 page table 的时候才 allocate 它

在 program 的一开始，我们只 allocate 了一个 first level page table

随着程序进行，我们需要的数据越来越多，每次需要新的 chunk of memory to hold data，我们就 allocate 一个 second level page table

因而，如果程序使用几乎整个 virtual address space，使用 multi-level page table 和使用 single-level page table，总共的 page table size 差不多，甚至 multi-level 更多一点（所有二级的 page table sizes 的和 等于同等的 single-level page table size，多出了一个一级 page table 的 size）

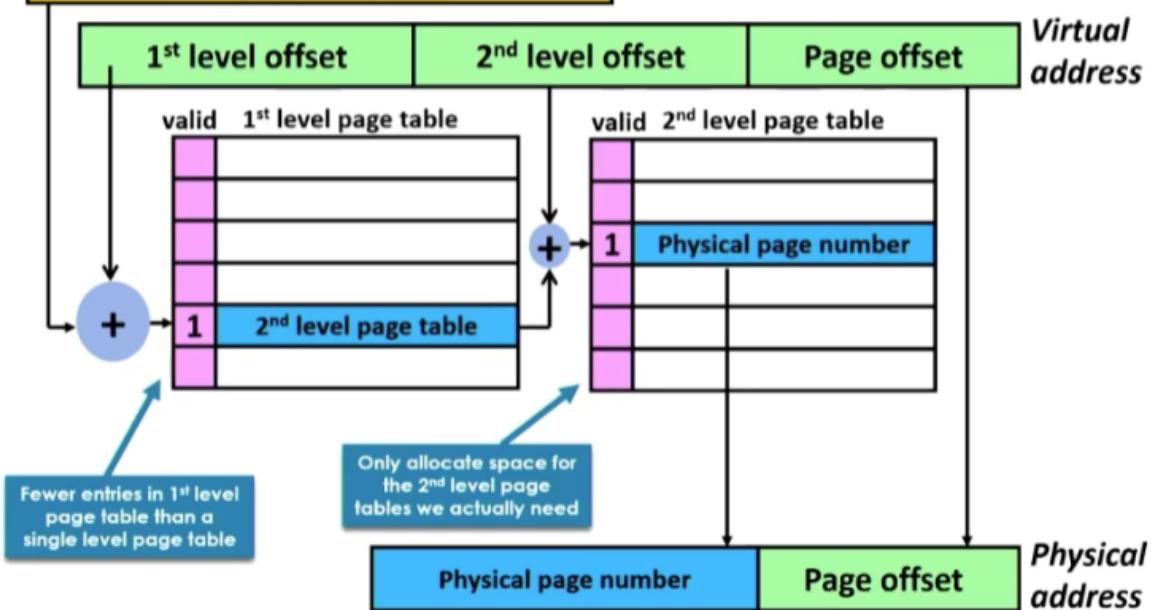
但是如果程序只用了 virtual address space 的一小部分，那么 使用 multi-level page table 的 总共的 page table size 就会远小于使用 single-level page table 的情况

Dividing an address

我们把一个 address 分为 1st level offset, 2nd level offset 和 page offset

Page table register

Top Level



Lec 23 - Speeding up VM

Table Look-aside Buffers (TLB)

我们现在的 VM 是比较慢的。

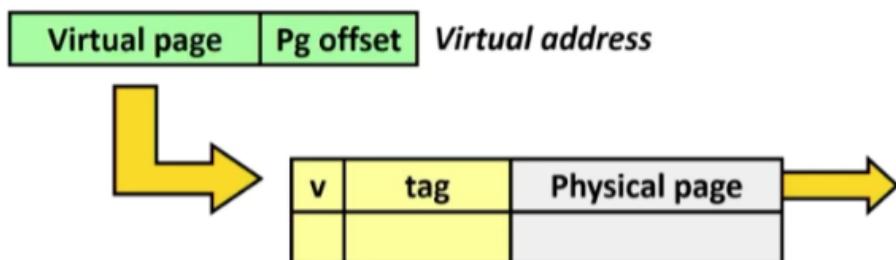
如果要 translate 一个 VM into 物理地址，对于 N-level page table，我们要在 getting the physical page number 前先 load N 次，然后我们再 access physical memory 以获得 data.

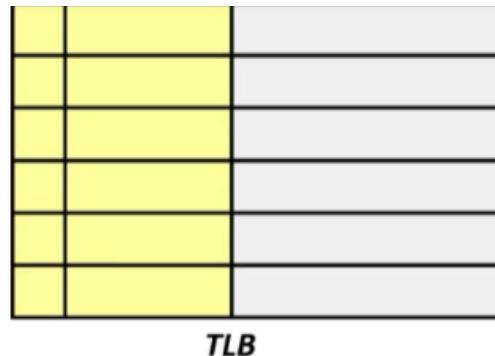
这太慢了，所以我们借鉴 cache 的 Idea:

TLB 是一个 cache of translations. 即：存储常用的 **virtual to physical address translation**，使得常用的 pages 可以跳过检索 N 次 page table，而是只需要检索一次就可以。

由于 Page table 的总用量其实不大，并且 spatial locality 根据 4KB 上下的 page size 肯定利用好了，TLB 的 miss rate 非常低。通常是 1%

一般设计 16~512 entries 的 TLB.





Virtual Memory walkthrough

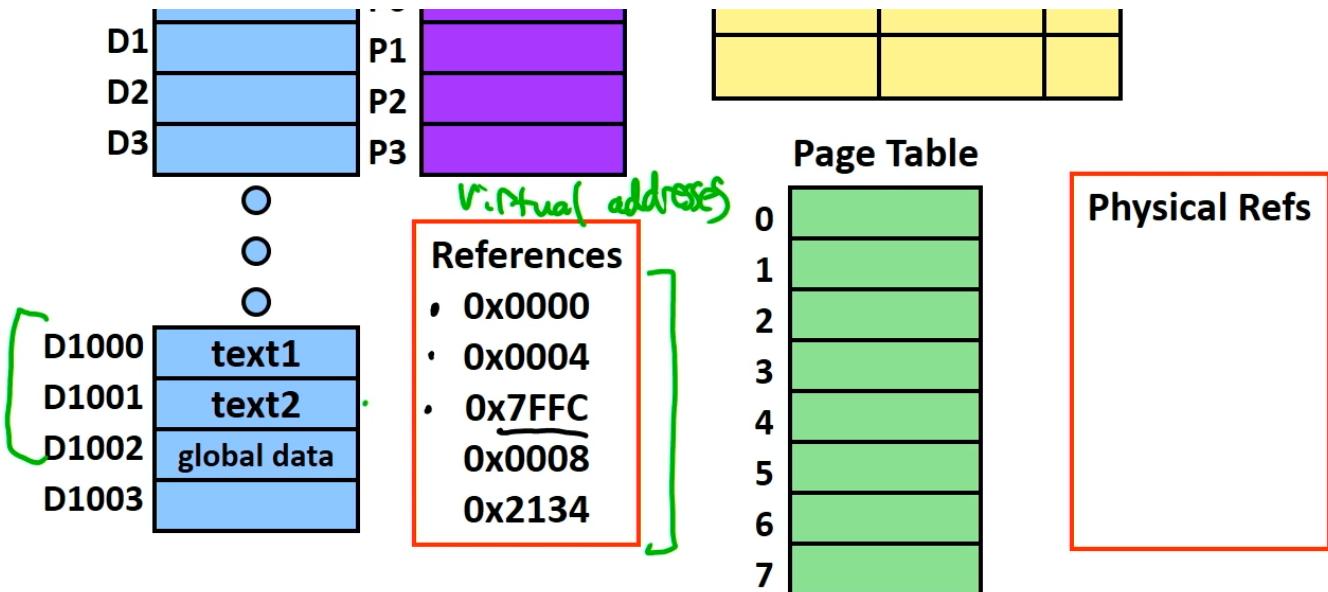
Load program into memory:

1. 让 OS 创建一个 new process
2. 为这个 new process 创建一个 page table
3. 标记这个 page table 里的所有 entries 都是 Invalid
4. 让 PTE (page table ptr) 指向 program 的 disk image, 即 exe 文件的头部所在的 physical page table

Page size = 4 KB, Page table entry size = 4 B

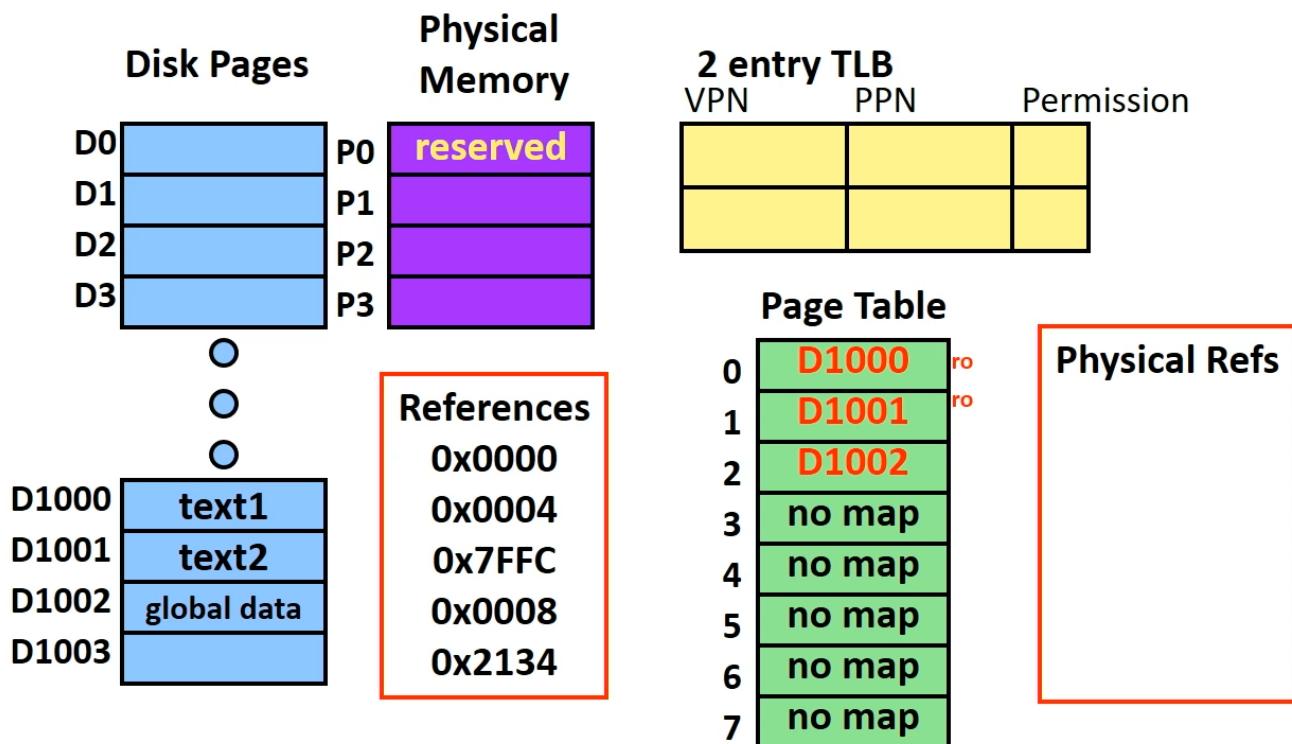
Page table register points to physical address 0x0000





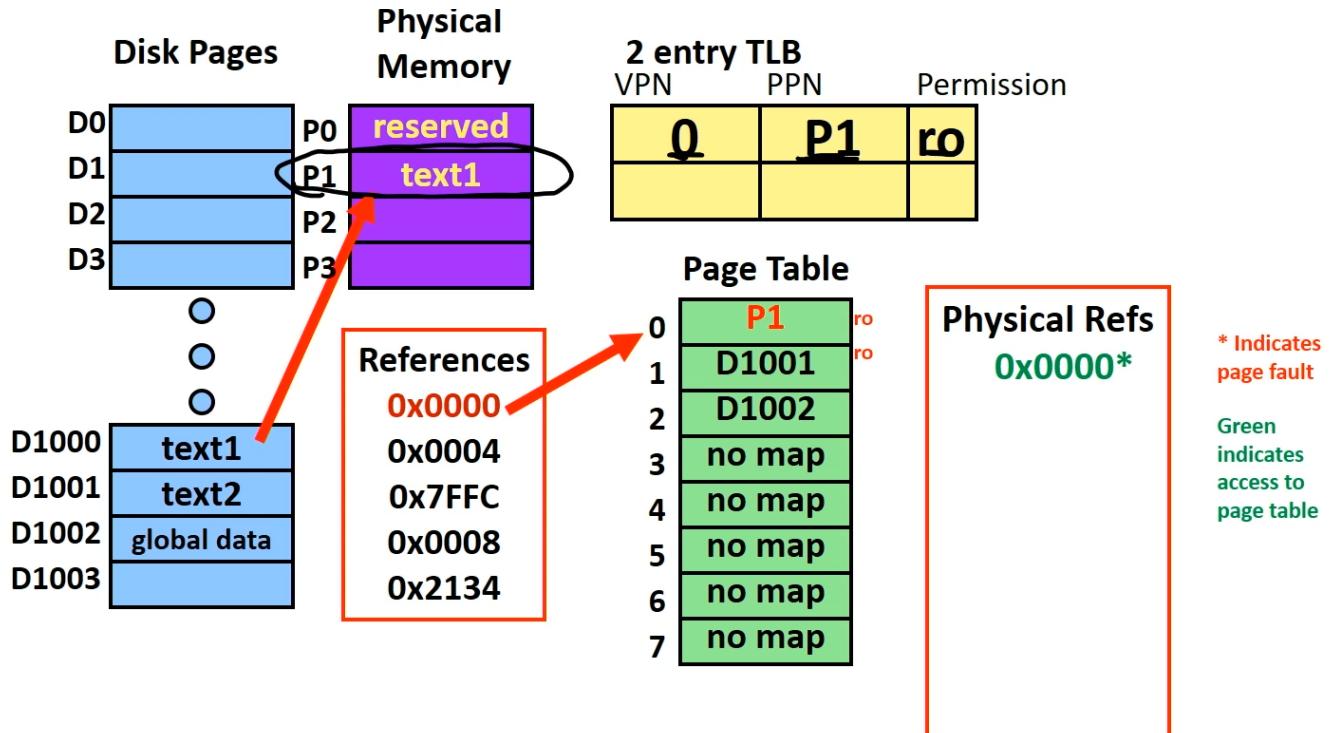
5. OS 会把 Physical memory 的开头的 blocks 留给 page table，并且让 Page table 的前几个 entries 用来指向 disk 上的 virtual physical page，映射到 text 和 data section

这个行为在随后会造成 immediate page fault，从而在读到对应 page 上的 instructions 时，会顺势和 physical memory 交换，把 text 转移到 physical memory 里

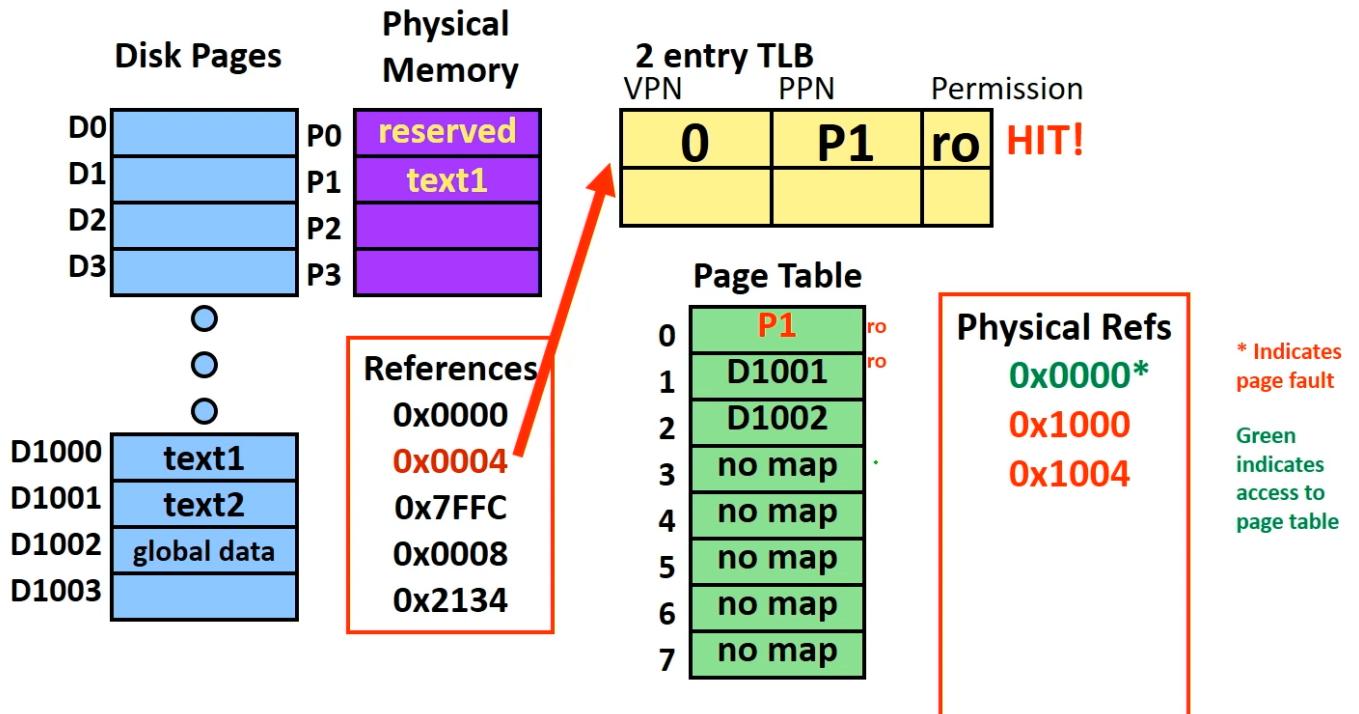


Run the program:

1. 第一个读到的指令在 virtual memory 0 的位置，我们已经在这里放上了会造成 page fault 的 disk pages 的引用。在读取 TLB 造成一个 miss 后，我们往 page table 上找，发现是 invalid 并且有值（说明在 disk 上），于是我们从它指向的 disk 位置，取对应 page 放进 memory 上的一个 page, say P1；同时我们取消这个 Invalid entry，把它替换成刚才 disk page 放入的 P1 位置；也就是我们把 Virtual 0x0000 翻译成了 Physical 0x1000



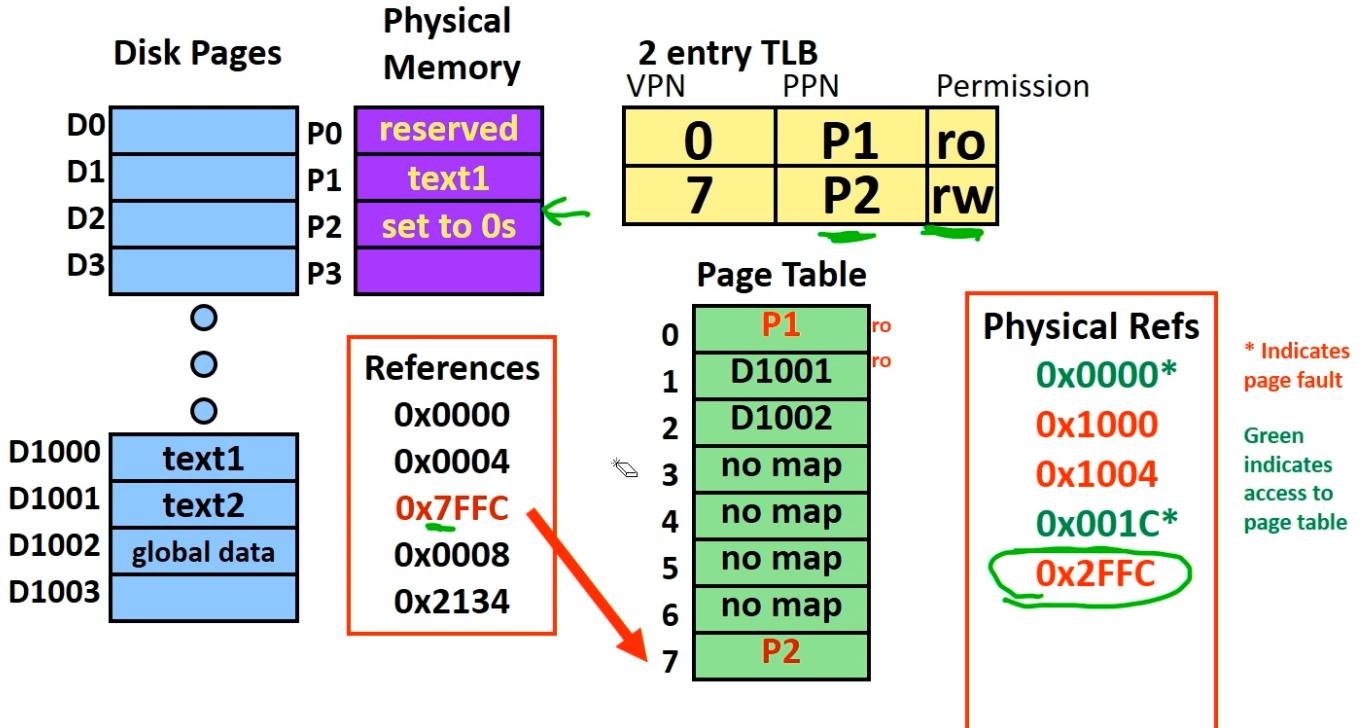
2. 读取下一个指令，是 0x0004，往 TLB 里查找，hit! 于是找到对应的 PPN P1，把它的 VPN 0 翻译成 PPN P1，把这个 VM 0x0004 对应到 PM 0x1004



3. 读取下一个指令 0x7FFC，对应 Page Table 的 entry 7，这应该是 stack 的 initialization，因为 stack 在 VM 的最底部。

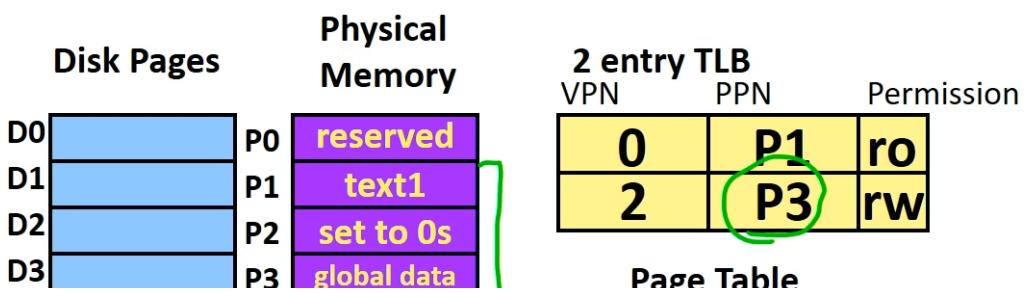
读 TLB, miss, 于是找一块 Physical Page 用来对应它, say P2, 于是把 Page Table 的 entry 7 更新为映射到 Physical P2

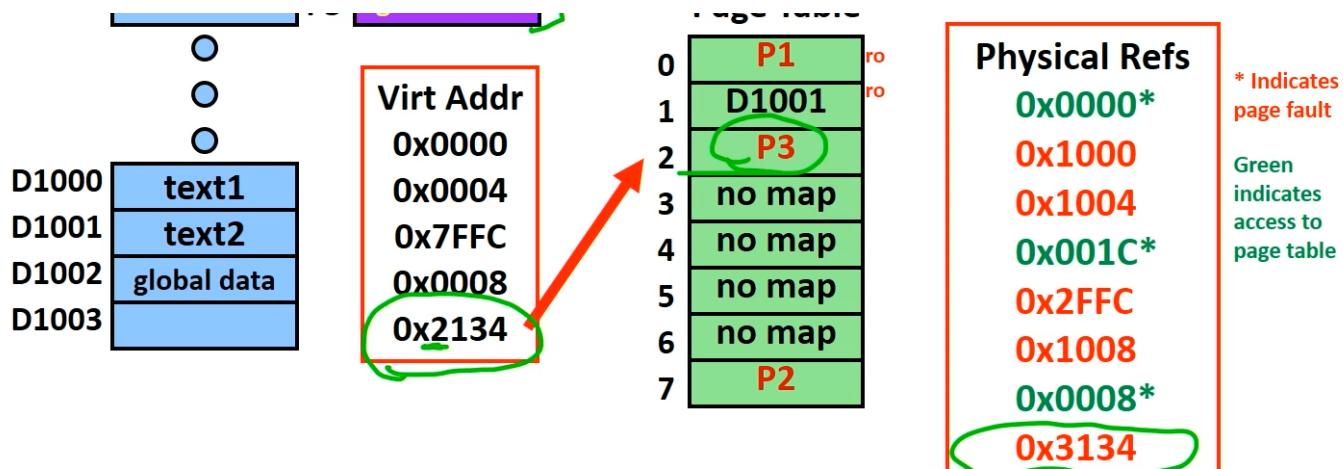
同时翻译这个地址：VM 0x7FFC 翻译为 PM 0x2FFC



4. 读取下一个指令 0x0008, trivial。

下一个指令 0x0008, 是读取数据的指令, 因为是在之前 initialization 的时候的 global data 应在的 page 上, 这会和我们读取第一个 0x0000 的指令一样造成一个 page fault, 于是我们从 disk 把 D1002 搬运到一个 Physical Page 上, say P3, 再把 Page Table 里 Virtual Page 2 指向 P3, 并完成翻译: 0x2134 翻译为 0x3134





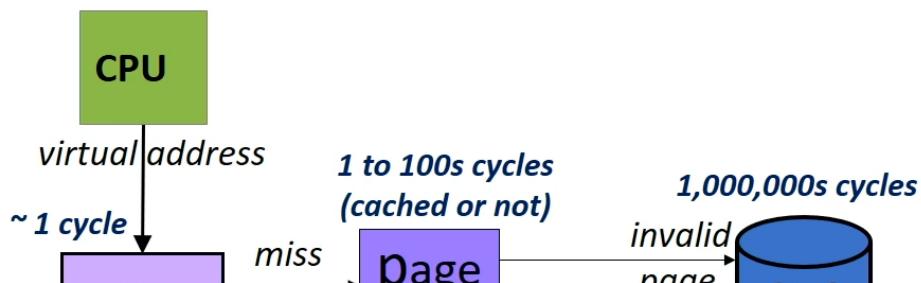
Placing Caches in a VM System

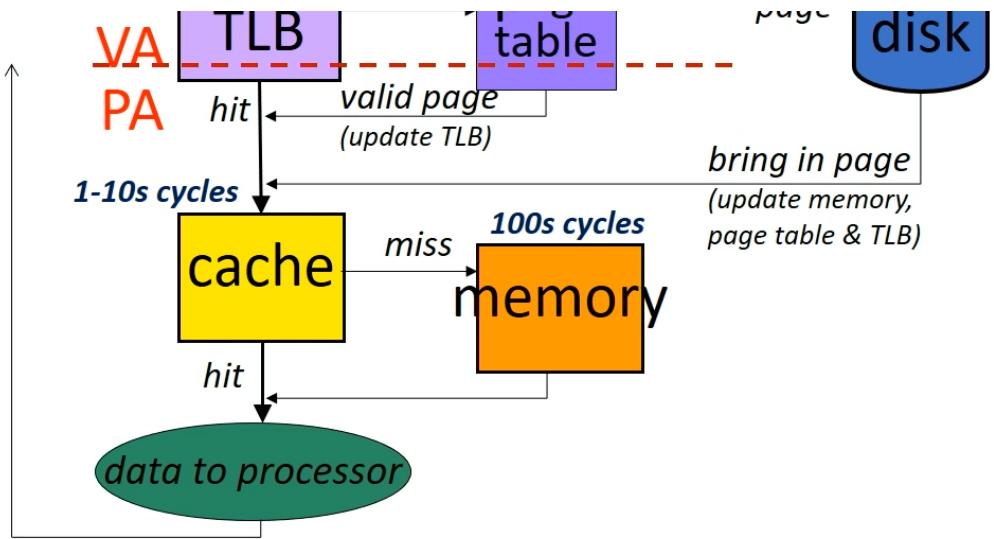
我们把 Cache 和 VM 结合起来，看整个 Memory Access 的流程

Question: Cache 应该 access Physical address (VM translation 后) 还是 Virtual Address (VM translation 前) ?

都可以，但是 Physical address (VM translation 前) 更慢；Virtual Address (VM translation 后) 更快

Physically addressed cache





Virtually addressed cache

