嵌入式-期末版

目录

6. Lecture 6 memory（内存）
    a. 内存种类
    b. Cache
    c. RAM原理
    d. 异常情况处理
    e. 内存和I/O映射
7. Lecture 7 peripheral communication（外围设备通信）
    a. 电磁学基础
8. Lecture 8 Architecture & Components （架构和组件）
    a. 冯诺依曼体系基础器件
    b. 行波进位加法器
    c. 进位选择加法器
    d. Booth乘法
    e. Carry propagate adder
    f. Carry save adder
9. Processor mode & Thumb code （处理器模式和 Thumb code）

1. 课程介绍 lecture 0
    a. assessment
        Final Exam. 2 hours, 40%。考试以往的不及格率：30%-40%
        Lab, 30%
        In-class Assignments, 30%。至少有60分
        只接受学校通知的请假
        如果及格，就不调分
2. 导论 lecture 1
    a. analogue to digital converters (模拟信号转数字信号，ADC)
    b. digital to analogue converters (数字信号转模拟信号，DAC)
    c. VDU：visual display unit，显示器
    d. 1024megabyte (MB）=1 gigabyte (GB)
3. Mnemonics（助记符） lecture 2
    a. 概论
        i. compiler把高级程序转成汇编或者是机器码，assembler把汇编码转成机器码
    b. 算术指令
        i. SUB rz, ry, rx： 表示z= y-x
        ii. RSB rz, ry, rx： 逆减，表示z=x-y
        iii. MLA rz, ry, rx, rw: Multiply and accumulate // y*x+w
        iv. BIC rz, ry, rx ： bit clear , ry AND NOT(rx)，x为1的地方都得为0
        v. 立即数要用在指令的最后一位
    c. 寻址
        i. 直接寻址
            1） 用寄存器的名字来寻址
        ii. 立即寻址（immediate address）

1) 使用立即数
2) MOV r12, #0xA0
3) MOV rZ, #N × $2^{2(16-M)}$ is 0xE3A0ZMNN
    a) N从0到255，M从0到15

iii. 间接寻址
1) LDR r6, [r11]：把 r11 的值作为地址，取得值放在 r6 上
2) 一个memory只有8比特，所以LDR和STR的地址使用这个地址以及它接下来的三位byte来运算

iv. Base plus offset addressing
i. LDR r6, [r11, #12]：用变化的副本寻址到内存中而本身不变
ii. LDR r6, [r11, #12]**!**：Pre-indexed，先变值，再寻址，有一个pling，!
iii. LDR r6, [r11], #12：Post-indexed，先寻址，再变值

d. Little endian 和 big endian
i. 小端就是least significant bit放在地址较低的位置上
ii. 大端则要注意，考虑的是byte，所以
1) 如果 STR r6, [r11]
2) 其中：
0XABCDEFGH in r6 and 0x00008000 in r11
3) 则0XGH 在 0X00008003

e. Load and store
i. LDRH、LDRB、STRH、STRB
1) 分别只存储加载半个word、byte

f. RISC的指令和高级语言相距甚远（high semantic gap），所以它的compiler也更复杂，Code density（同等体积的代码的信息量）低

4. Flag（标志位） Lecture 3
a. 概论
i. flag也称为condition codes，状态码
ii. ARM7的flag
1) zero flag (Z)
2) negative flag (N)
3) carry flag (C)
4) overflow flag (V)

iii. 所有的ALU指令都有S后缀版，表示运行这行代码时要设置标志位：MOVS, ADDS, SUBS, RSBS, MULS, MLAS, ANDS, EORS, ORRS, BICS
iv. 每一次四个flag要么被set为1，要么被reset为0，完全不管它们原来的值是多少

b. conditional execution
i. 几乎所有的ARM指令都有conditional execution版本，这由指令最后的两个字母决定
1) MOVCS r12, #114：如果carry flag是set
2) 重要的：EQ、NE、CS、CC

15 different condition fields

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

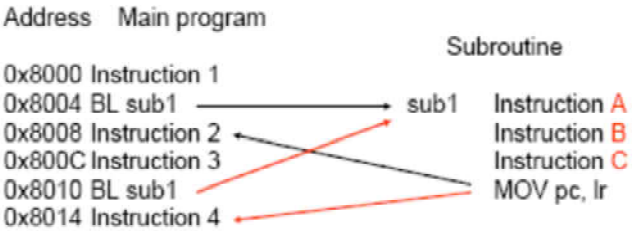
c. conditional branch:
   i. 直接跳转到某个位置
      1) 例子：BNE 0x08C00000
      2) 只有24bits可以用来作为地址
      3) 但是地址的末四位一定是00，不然就会不对齐，所以可以表示的范围是± 32MB （2^20Byte=MB，2^(4+2-1)=32）
   ii. 跳转到子程序（subroutine）
      1) 使用label
      2) 例子：BNE label2

         .....

         label2 ADD r5, r4, #76
   iii. ADC
      1) 比如：ADC r0, t1, #3 是将 r0和3之和外加carry flag放入 r5
d. 二进制补码 lec3
   i. 略。

5. 浮点数 lec3
   a. 只考虑单精度浮点数
      i. 由 IEEE754 规定
      ii. 对于32位浮点数：1位符号位 + 8位指数位 + 23位小数位
      iii. 记得小数位要正规化
      iv. 为了表示正负exponent，bias= 127. 你写上去的指数数字得是你要的数字+ 127
      v. 特殊case
         1) 0：exponent和fraction都是0，忽略符号位 1.0 * 2^000000000000
         2) 正负无穷：exponent全是1，fraction全是0 1.0000000*2^1111111
         3) NaN：exponent全是0，但是fraction不是全0 1.21 *2^11111111
         4) 非常小（小于 1.17*10^-38）：exponent是0 1.23* 2^00000000
6. subroutine（子程序） lec4
   a. 表示
      i. r14是link register，在mnemonic中写作lr

ii. branch & link：写作 BL



b. nested subroutines

i. 如果在函数中调用函数，则 lr 的值需要被存储到stack中

ii. 这个stack的bottom是固定的，但是它的top是不固定的，栈顶存储在r13中，又称 stack pointer，即sp

iii. 栈顶的值要么表示最后一个元素（这种模式叫做full stack），要么表示最后一个元素上面的那个空位置（这种模式叫做empty stack）

iv. 要么栈顶元素的地址不小于栈底：ascending stack；要么descending stack

c. 调用stack的方式

i. 通常调用函数时，为了将lr的值更新，第一个命令是（在descending stack情况下）：STMFD sp!, {lr}

ii. pop回lr：LDMFD sp!, {lr}

1) 然后就将值移入pc： MOV pc, lr

2) 这两条指令合在一起等价于 LDMFD sp!,{pc}

iii. 可以一下子将多个寄存器的值移入移出栈中

1) STMFD sp!, {r6-r9, lr}

a) 先存储lr，再存储 r9，……

2) LDMFD sp!, {r6-r9, lr}

3) 注意顺序

Order for Store: r15, r14, …, r0

Order for Load: r0, r1, …, r15

d. barrel shifter lec4

i. LSL/LSR:逻辑左、右移

ii. ASR：算术右移，补符号位

iii. ROR：rotate right：循环移动，低位补高位，高位补低位

iv. RRX：rotate right extended：右移一位，用carry flag补最高位，并且把原本最右边的位移入carry flag

v. MOV r1, r2, LSL #5

1) 先shift，再加，再move

vi. 用位移的方式表示乘法

1) *64： MOV rx, ry, LSL #6

2) *17： ADD rx, ry, ry, LSL #4

7. Interrupt（中断） lec5

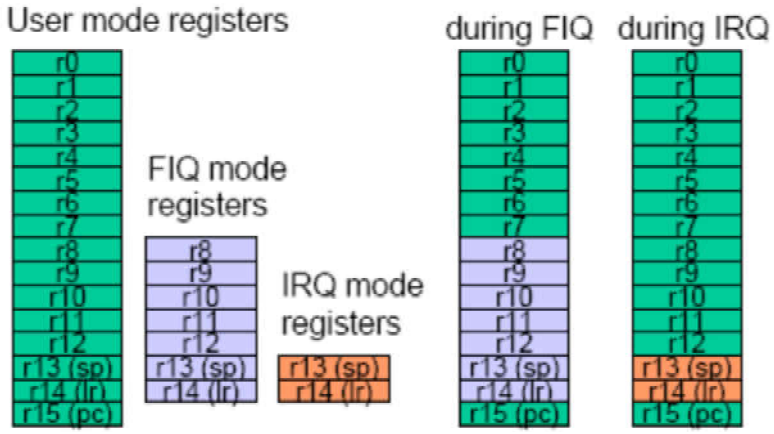
a. ARM7有两种中断

i. normal interrupt (IRQ)

ii. fast interrupt (FIQ)

b. 不同模式下能够使用的寄存器是不一样的

i. 每种模式都有其link register 和 stack register，并且FIQ还有它自己的r8 - r12



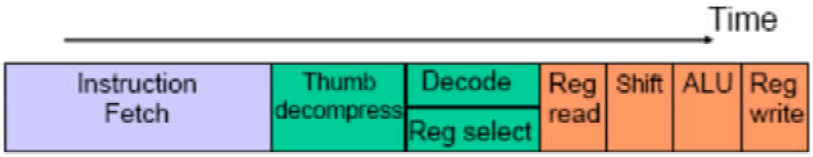
c. Current Program Status Register （CPSR）
   i. 用来存储当前的condition code bit
   ii. 程序员使用它的时候不需要考虑其底层实现
   iii. Current program status register（当前程序状态寄存器，CPSR）
      i. 组成成分
         1) 状态码：4比特：N、Z、C、V
         2) 当前状态：5比特：User, FIQ, IRQ, SVC, Abort, Undef and System
         3) 中断：2比特：I、F
         4) Thumb比特：1比特：T

mode	4	3	2	1	0
User:	1	0	0	0	0
FIQ:	1	0	0	0	1
IRQ:	1	0	0	1	0
SVC:	1	0	0	1	1
Abort:	1	0	1	1	1
Undef:	1	1	0	1	1
System:	1	1	1	1	1

d. 异常处理
   i. 处理异常前
      1) IRQ或者FIQ模式的寄存器被激活了
      2) CPSR被存入SPSR（Saved Program Status Register）
         a) SPSR中有两个寄存器，分别对应IRQ和FIQ
      3) return address被写入相应模式的link register
      4) 将0x00000018（IRQ时） or 0x0000001C （FIQ时）写入PC，让程序跳转到相应位置
   ii. 处理过程
      1) 首先执行的是在内存地址0x00000018（ IRQ时） or 0x0000001C （FIQ时）的指令
      2) 对于IRQ，0x00000018只能指向另一个内存地址，因为它后面的指令是属于0x0000001c的
      3) 对于FIQ，则不必如此，因为它后面的地址可以被直接使用
   iii. 处理之后的恢复

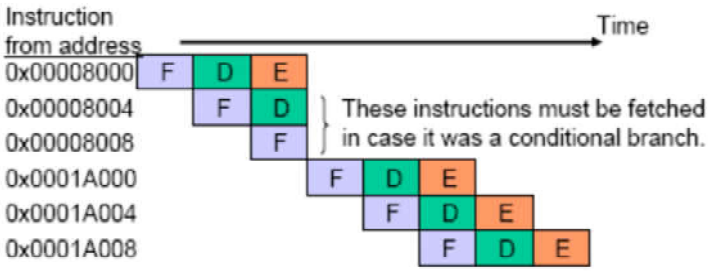
1) SPSR被复制回了CPSR
2) IRQ和FIQ中的link register被复制进了PC

8. instruction pipeline（流水线） lec5
   a. 具体而言

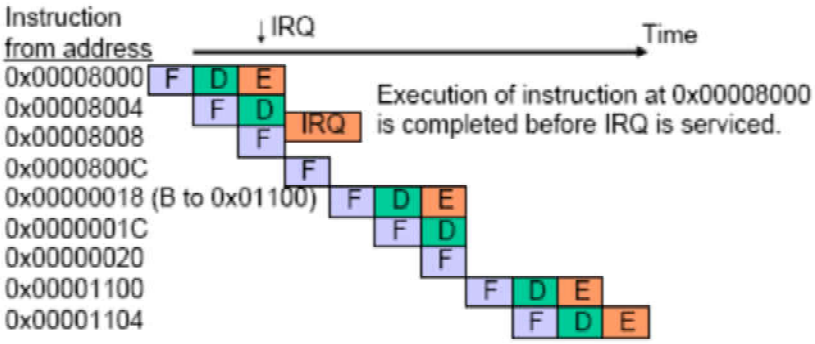


   i. **FETCH** Instruction fetched from memory. **DECODE** Thumb to ARM decompression. Decode. Register selection **EXECUTE** Register read. Barrel shift. ALU operation. Register write.

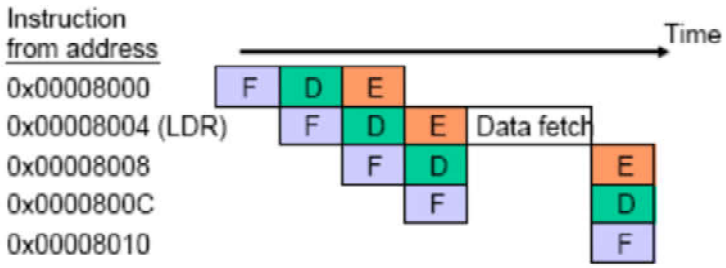
   b. 优化操作
      i. 如果指令都是在连续的内存中并且没有数据冲突，CPI就为1
      ii. 如果有branch、load、interrupt，就无法取得最好的performance
   c. branch
      i. assume that the instruction at address 0x00008000 is Branch to 0x0001A000



         1) 如果一个conditional branch没有执行，那么就没有clock cycles丢失
   d. IRQ

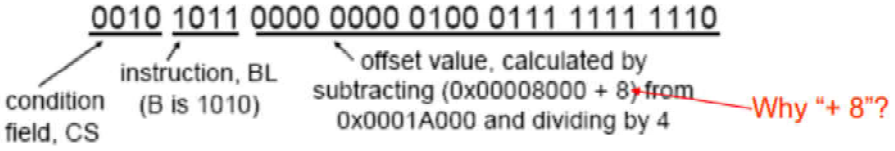


   e. interrupt latency
      i. FIQ至少需要4个时钟周期，因为要跳转过去再跳转回来
         1) FIQ可能会被system reset中断
      ii. IRQ至少需要7个时钟周期，因为要跳转两次
         1) 同时IRQ还有可能被FIQ中断
   f. 如果要取数据



g. eliminate bus conflict
   i. 冯诺依曼结构中，数据和指令都在一个bus中获取，这样可能会有冲突
   ii. 哈佛结构则不会，哈佛结构中数据和指令在两个bus中获取，但是哈佛结构更加复杂

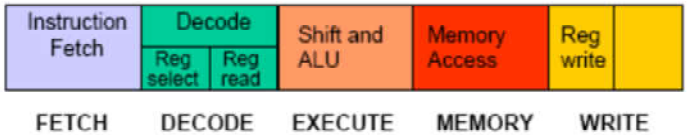
h. Branch
   i. BLCS label
   ii. 假设当前的位置是0X8000，而label在 0X1A000 ，则指令应该是：



   iii. 为什么是 +8 呢？Execute 0X8000这条指令时，解码的指令是0X8004，PC 中fetch的值是0X8008

i. ARM 和 传统的 RISC 的区别
   i. ARM 不是 每个周期只有一条指令
      1) 比如：STMFA r13!, {r0-r2, r14} 需要5个周期来execute
   ii. ARM 可以直接在内存层面操作数据
   iii. ARM 不能延迟 branch
   iv. 寄存器数量少，只有 16 个

j. 性能提升方式
   i. 更多更长的 pipeline
      1) 3 stage or 5 stage?
         i. 如果选择3步，则每一步步长大，时钟频率低
         ii. 但是5步的话，功耗大，因为在一步中同时执行的circuit多
      2) 在ARM7 的三步架构中，瓶颈是execute，而五步就可以分散execute的压力



   ii. 使用哈佛结构
   iii. 延迟 branch
   iv. 再加一些有特殊用途的（specialized）指令

k. Interlock
   i. ARM9采用了5 stage，因此在对一个寄存器memory access时，下一步不能立即使用这个寄存器，要锁住

9. Memory（内存） lec 6
   a. silicon ROM种类
      i. Programmable ROM (PROM)
         1) 只能编一次

ii. Erasable PROM (EPROM)
    1) 在 ultraviolet (UV) light 光照下可以抹去，并且重新编程
iii. Electrically erasable PROM (EEPROM)
    1) 在电信号下可以抹去重写
iv. Flash Erasable PROM (FEPROM or flash memory)
    1) 和EEPROM很像，但是只有一部分可以被擦除

b. RAM
    i. DRAM
        1) 需要不断刷新
    ii. SRAM
        1) 不需要刷新，但是需要更多的硅片表面积，所以更贵

c. cache
    i. temporal and spatial locality
    ii. hit rate & miss rate
    iii. mean access time计算
        1) h表示hit ratio，m表示miss ratio
        2) 理想情况下：

$$t_{ave} = h \times t_c + m \times t_m = h \times t_c + (1-h) \times t_m = (1-m) \times t_c + m \times t_m$$

        3) 实际情况下要加一个额外时间a（cache control and routing circuits）：

$$t_{ave} = h \times t_c + m \times t_m + a = h \times t_c + (1-h) \times t_m + a$$
$$= (1-m) \times t_c + m \times t_m + a$$

    iv. cache设计
        1) 两种设计方案
            a) associative cache
            b) direct mapped cache
        2) 计算
            a) 假设一个block中有 X=2^a 个bytes，然后这个内存是Y位的，所以cache的 block number 有 Y-a 位
            b) 要有 valid bit
            c) 所以一个block有 1+ Y-a + 8*2^a 个bit
        3) associative cache
            a) 在fully associative cache中，为了找有没有这个tag都要花时间遍历所有tag；即使不是fully，其实也要花时间在遍历上
            b) 如果使用associative cache，就要考虑更换相应位置的block的方式
            c) tag field中只会标注前几位，如果你想具体地找到某一位byte还得把这个block content 当成数字，取index
            d) 比如这个D4的地址其实是：0x37B6A038

Tag field	Block contents
0x9AB6301	0xC2A1096C 6FE3D674 A956410B D4FE3D67
0x54B09FF	0xD4FE3D67 A956410B 6FE3D674 C2A1096C
0x37B6A03	0x6FE3D674 C2A1096C D4FE3D67 A956410B
0xF31A004	0xC2A1096C D4FE3D67 6FE3D674 A956410B
0xFF4E502	0x6FE3D674 C2A1096C A956410B D4FE3D67
0x07D3A00	0xA956410B C2A1096C 6FE3D674 D4FE3D67
…	… … … …
0x96301AB	0x6FE3D674 C2A1096C D4FE3D67 A956410B

0 8 15

4) Direct-mapped cache

Cache line no.	Tag field	Block contents
0x00	0x07D3A	0xC2A1096C 6FE3D674 A956410B D4FE3D67
0x01	0x9AB63	0x6FE3D674 C2A1096C D4FE3D67 A956410B
0x02	0xFF4E5	0xC2A1096C 6FE3D674 A956410B D4FE3D67
0x03	0x37B6A	0x6FE3D674 C2A1096C D4FE3D67 A956410B
0x04	0xF31A0	0xC2A1096C 6FE3D674 A956410B D4FE3D67
…	…	… … … …
0xFF	0x54B09	0x6FE3D674 C2A1096C D4FE3D67 A956410B

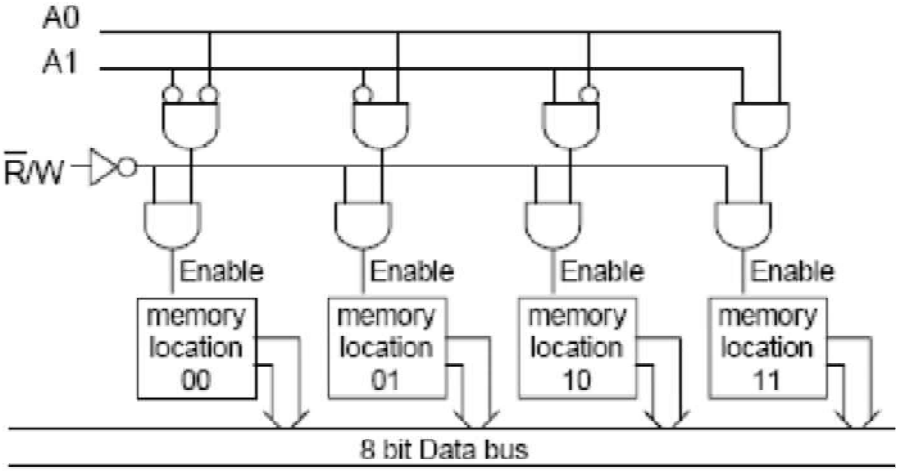
0 8 15

d. 数据写入
   i. 问题：如果只写入cache不写入内存，要考虑内存没有更新的情况
   ii. 解决方案
      1) Write through
         a) 每次写入cache时都写入内存
      2) Copy back
         a) 在cache中加一个hit bit,表示是否被hit过
         b) 平时只写入cache，在替换这个cache时就看看hit bit，如果被set了，就在替换时写入内存

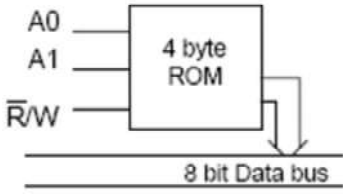
e. Tri-state gates
   i. 不能有多个设备同时使用bus
   ii. 解决方法：加一个enable 开关
      1) 表示0、1、high-impedance（enable=0时）

enable	input	output
0	0	high-impedance
0	1	high-impedance
1	0	0
1	1	1

f. RAM原理
   i. 控制内存读取

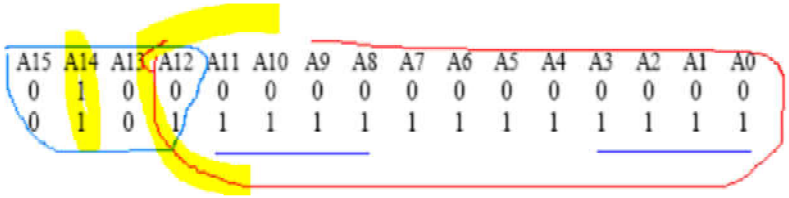


简化：

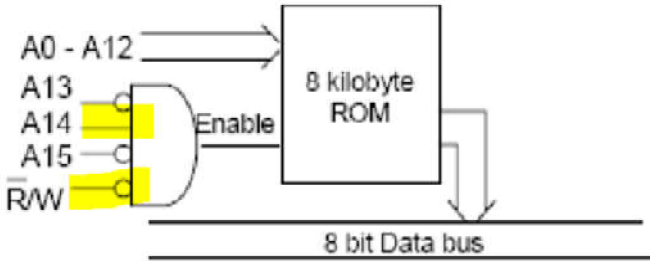


g. 异常情况处理

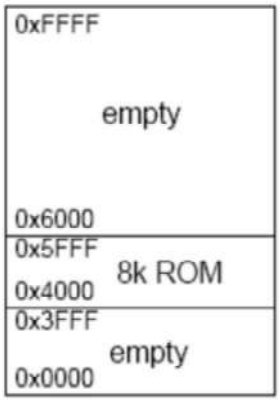
i. 如果内存只有13位（8kB），而bus有16位：



1) 等价于使用多余的几位bit来控制enable开关



等价的内存结构：



h. Memory mapped I/O

i. 把硬件中的IO和内存map一下，这样，如果相应的内存有变化，则硬件中的Output如led等也会有变化；如果硬件的input有变化，内存中也会有相应的变化

ii. 大概会考

1) 控制IO以及代码写法及解释

Example 1: sequential accesses to IOSET and IOCLR affecting the same GPIO pin/bit

State of the output configured GPIO pin is determined by writes into the pin's port IOSET and IOCLR registers. Last of these accesses to the IOSET/IOCLR register will determine the final output of a pin.

In the example code:

```
IO0DIR = 0x0000 0080 ;pin P0.7 configured as output
IO0CLR = 0x0000 0080 ;P0.7 goes LOW
IO0SET = 0x0000 0080 ;P0.7 goes HIGH
IO0CLR = 0x0000 0080 ;P0.7 goes LOW
```

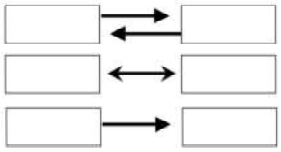
pin P0.7 is configured as an output (write to IO0DIR register). After this, P0.7 output is set to low (first write to IO0CLR register). Short high pulse follows on P0.7 (write access to IO0SET), and the final write to IO0CLR register sets pin P0.7 back to low level.

10. Peripheral Communication （外围设备通信） lecture7
    a. 电磁学基础

    if the input is between 0 and V_{IL} , it is considered a low.
    If the input is between V_{IH} and V_{DD} , it is considered a high.

    i. $V(t) = 3.3 - 3.3e^{-t/RC}$
    ii. $C = \varepsilon_0 \frac{A}{d}$

    b. 设备驱动器是各种软件功能的集合，用来让高层次的软件使用I/O设备

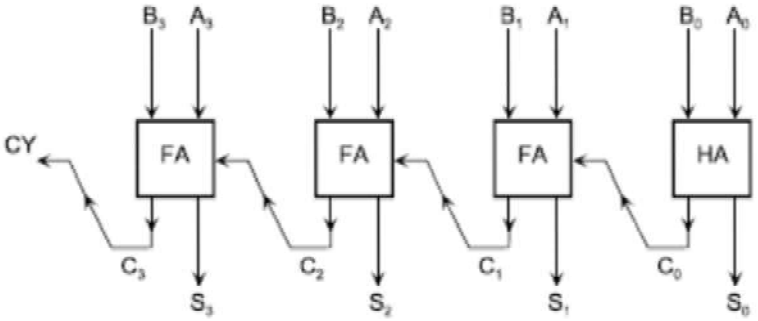
    i. 

    c. 通信
        i. 串行通信时，两边的bit rate必须相同
        ii. 方式
            1) 种类1：polling：处理器不断地检测看是否有通信任务要完成
            2) 种类2：interrupt：
                a) 方式1：处理器给peripheral一个信号，表示要开始通信了，然后等待通信完成的信号
                b) 方式2：处理器等待peripheral的通信任务请求的interrupt
        iii. device driver设计哲学
            1) 封装复杂性
11. Architecture & Components （架构和组件） lecture 8
    a. 概论
    b. 计算机体系和计算机架构的区别
        i. computer architecture
            1) 从用户的角度来看待，包括指令集、内存管理、中断处理等
        ii. computer organization
            1) 从用户不可见的角度来看待，包括pipeline结构、各级缓存等
    c. 冯诺依曼体系基础器件
        i. 存储data和instruction的内存
        ii. control和decoder
        iii. arithmetic & logic unit（ALU）

iv. I/O设备

d. 行波进位加法器

i. 第一个是HA，half adder



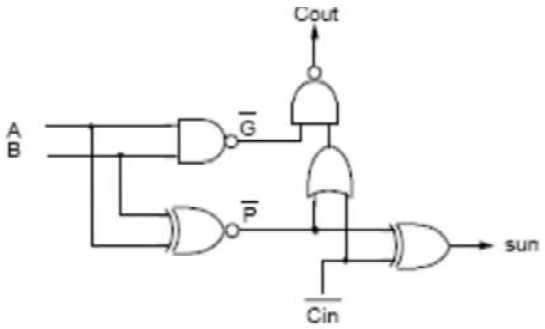
ii. 圆圈里面加上加号表示亦或、. 表示且

A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$Sum = (A \oplus B) \oplus Cin$$

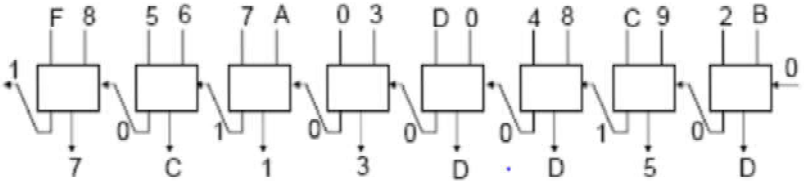
$$Cout = A.B + A.Cin + B.Cin$$

iii. 由此构建出的结构

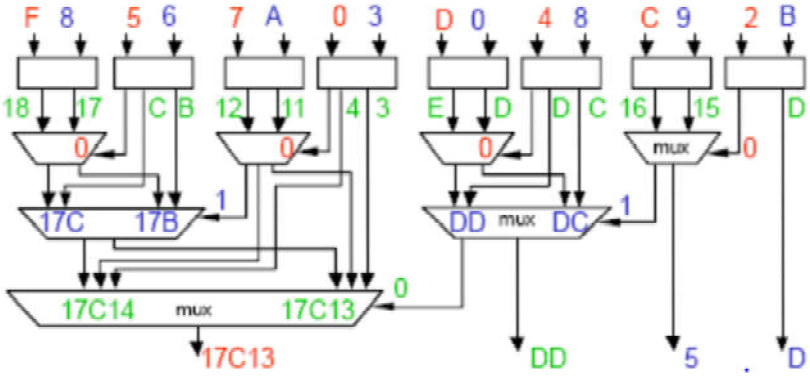


$$Cout = \overline{\overline{G}.(\overline{P} + \overline{Cin})}$$

iv. 4-bit加法器，每一位都由4比特构成



v. Carry select adder



vi. performance比较（需要的时钟周期时延）

Size of adder	Ripple carry	Look ahead	Carry select
4 bits	4	1	1
8 bits	8	2	1
16 bits	16	4	2
32 bits	32	8	3
64 bits	64	16	4

e. Booth 乘法

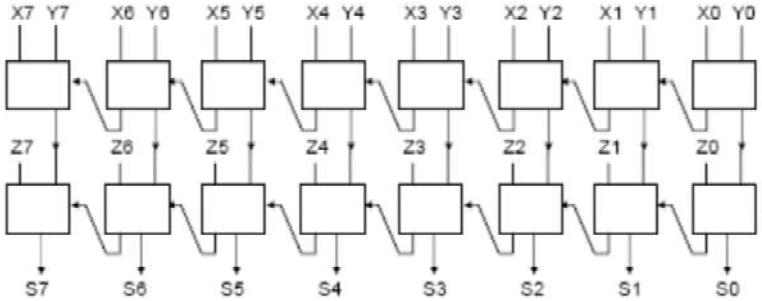
i. 过程

1) 将乘数（multiplier）放入寄存器M中
2) 设进位为Cin
3) 设最终结果是T
4) 循环以下过程
   a) 设这是第N个循环
   b) 取M的最低两位数，称之为B，与Cin相加，判断：
      i) 如果Cin+B=0，则设Cout=0
      ii) 如果Cin+B=1，则左移被乘数2N位，把它加入T，令Cout=0
      iii) 如果Cin+B=2，则左移被乘数2N+1位，然后把它加入T，令Cout=0
      iv) 如果Cin+B=3，则左移被乘数2N位，然后从T中减去它，令Cout=1
      v) 如果Cin+B=4，则设Cout=1
   c) 令Cin=Cout
   d) 右移M两位

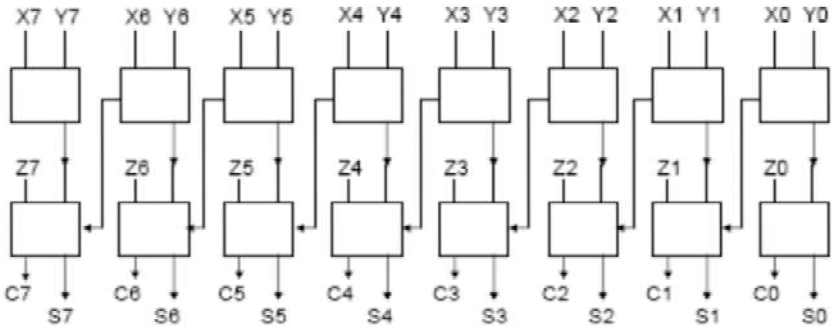
ii. Booth乘法可以作用于负数，只要使用二进制补码系统就可以了（包括计算过程中如果出现了负数也是这样处理的）

iii. 效率：对于N位的运算，只需要N/2的时钟周期

f. Carry propagate adder



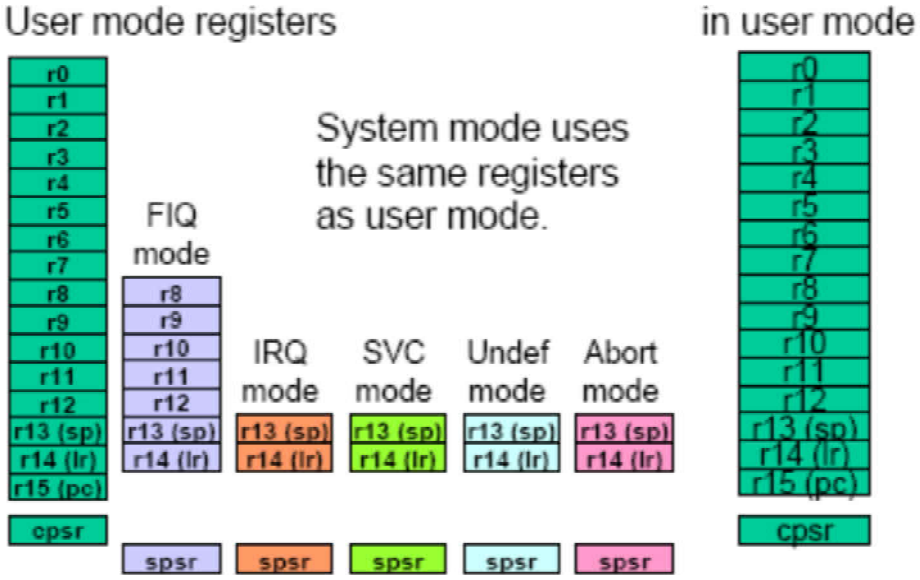
g. Carry save adder



12. Processor mode & Thumb code （处理器模式和 Thumb code） lecture 9

a. saved program status register (SPSR)

i. 用来在改变了CPSR时保存CPSR，恢复后再拷贝回去
ii. SVC：supervisor模式，软件中断时使用
iii. abort：当读写内存违规时使用
iv. Undef：instruction decoder发现没有机器码可以decode时使用
v. system：OS的特权模式



b. Thumb 指令集
i. 16位，是经过压缩的ARM码
ii. 并非所有的ARM码都有Thumb码
iii. Thumb de-compressor
1) 使用Thumb的模式时，每一个word都会被拆分成两个半word
2) 其中一个包含这条指令的内容，在本周期执行
3) 另一个在下一个周期被解码以及执行
iv. 和ARM指令的比较
1) Thumb码在16位存储系统中更快
a) 所以如果一个系统看重能耗和内存占用，则设计上会偏向于选择16位存储系统和Thumb
2) ARM码在32位的bus上更快
a) 所以如果一个系统看重能耗和内存占用，则设计上会偏向于选择32位存储系统和ARM
3) 一般系统二者兼用，ARM用来处理重要的事务，Thumb用来处理后台任务