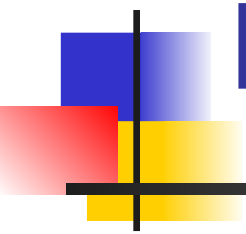
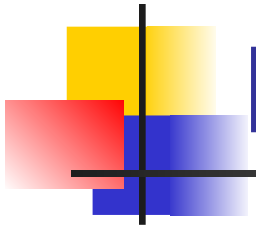


PMON 的解读和开发



PMON 的发展和编译环境

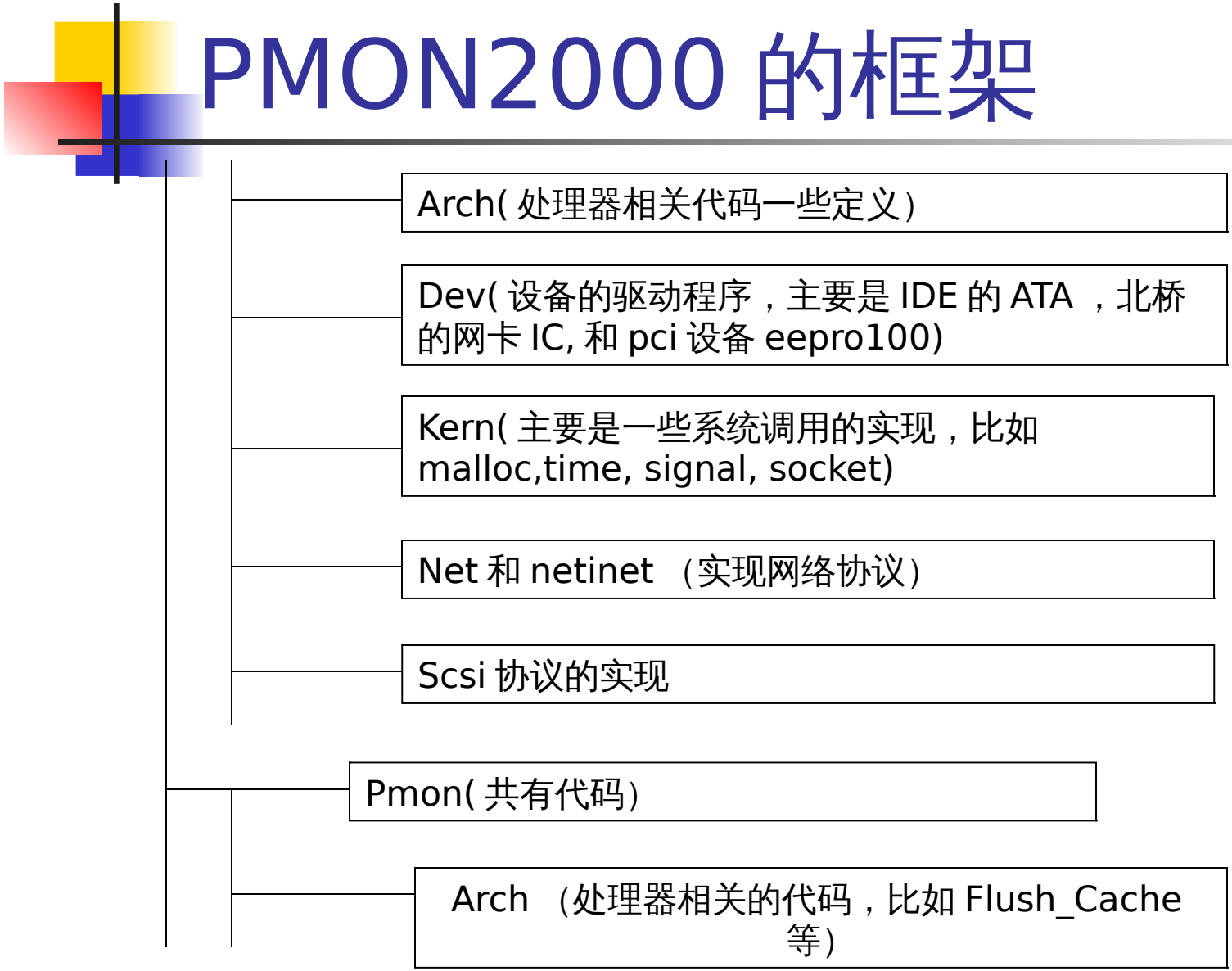
- PMON
  - PMON 的早期版本的功能有： shell, net, load, debug 。不支持硬盘，显卡。并且扩展性不好，编译器是 sde-gcc.

PMON 的发展和编译环境

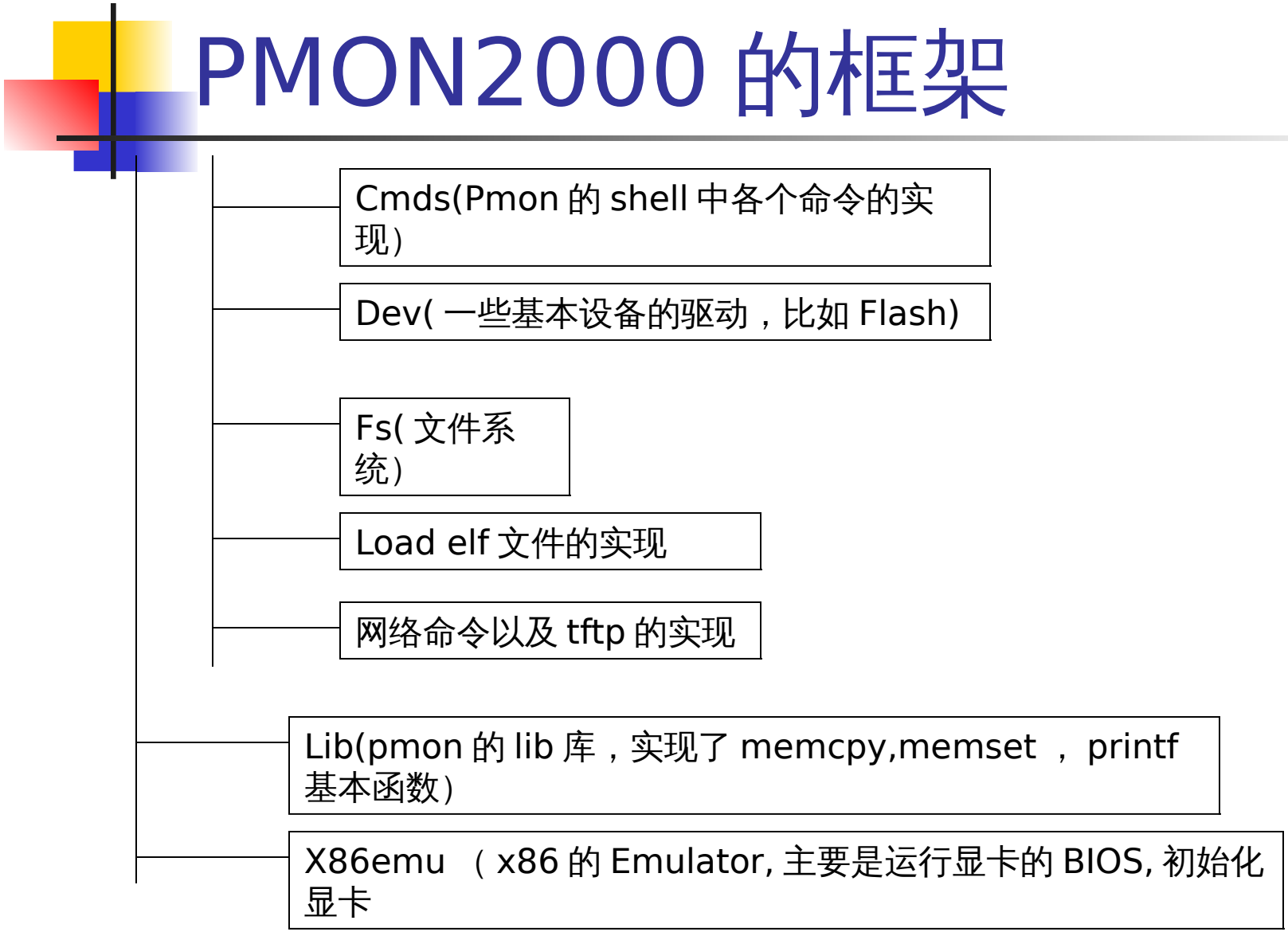
- PMON2000
  - 现在龙芯 1 和 2 用的 BIOS，在原来的 PMON 的基础上添加了硬盘支持，文件系统 ext2 和 fat 的支持，显卡的支持等等。修复了 debug 功能，扩展性也得到提高。比较容易移植到新的系统。
  - 编译器为 mips-elf-gcc

PMON2000 的目录结构

PMON2000 的框架



PMON2000 的框架

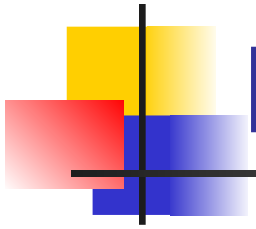


Targets 目录的组成

- 每个系统一个目录，我们拿 Bonito 来为例子，主要有下列文件：
  - start.S 位于 Targets/Bonito/Bonito 目录下，是 C 环境建立之前的汇编代码，使整个 BIOS 运行的起点。
  - tgt_machdep.c 位于 Targets/Bonito/Bonito 目录下，一些板子相关的函数。
  - pci_machdep.c 进行 Targets/Bonito/pci 空间分配的一些函数
  - Targets/Bonito/dev 目录下一些板子特殊的设备的驱动。
  - Targets/Bonito/conf 目录下是一些编译环境建立需要的一些文件

PMON 编译环境的建立

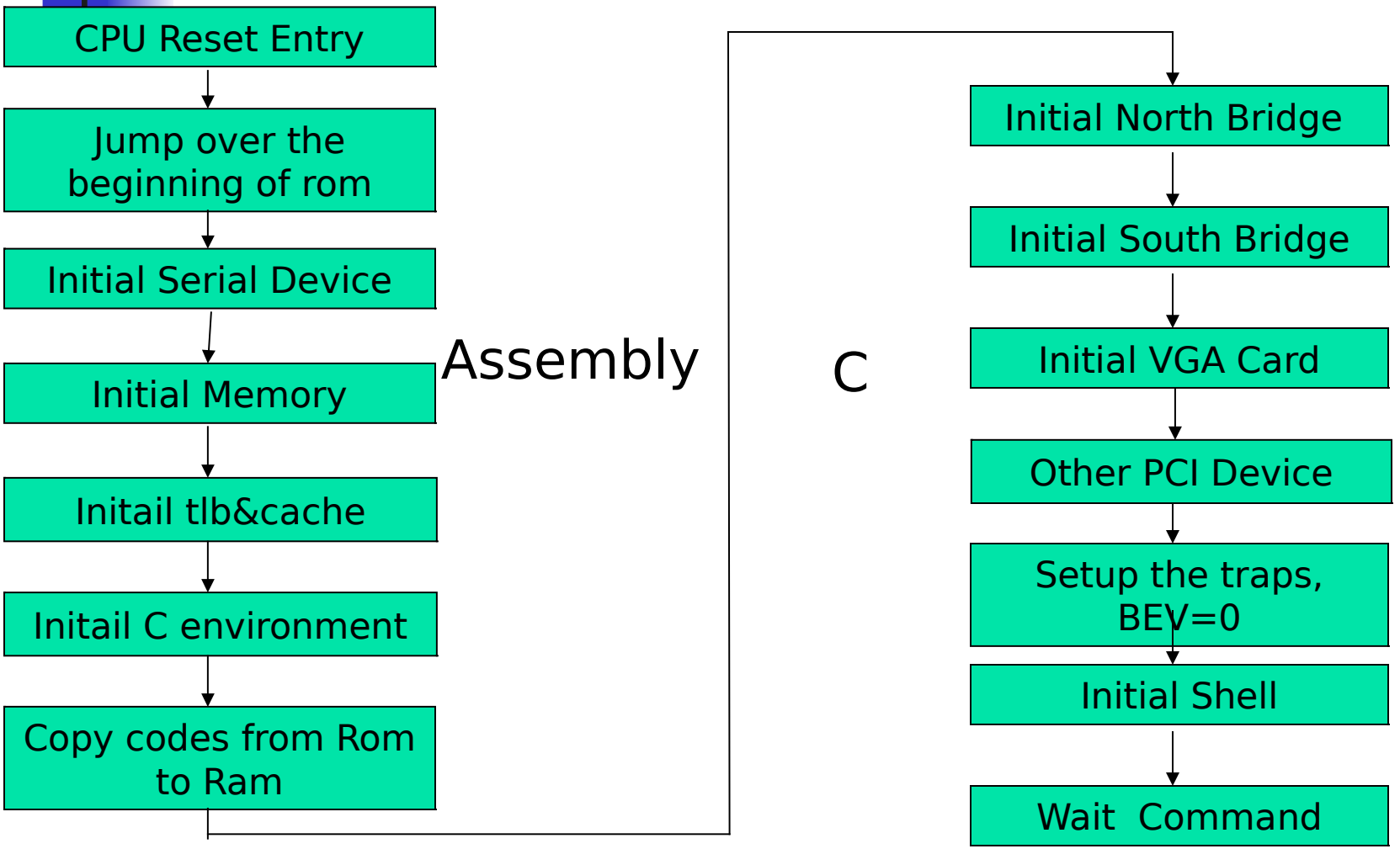
- 将 comp.tar.gz 在 / usr/local 解开
- 将 / usr/local/comp/mips-elf/gcc-2.95.3/bin 加入到 PATH 目录下
- 进入 pmon2000 的 tools 目录下 make ，建立一些 conf 需要的工具。
- 进入 pmon2000 的 Targets/Bonito/conf 目录中
- 编辑 conf 目录下 Bonito 文件，选择需要编译的模块
- tools/pmoncfg/pmoncfg Bonito(conf 类型文件）。形成目标主目录下的 compiler 目录
- 进入 Targets/Bonito/compiler/Bonito 的目录， make 形成 pmon 。



PMON 编译环境的建立

- Makefile 是根据 Targets/Bonito/conf/Makefile.Bonito 文件形成的。
- 链接脚本是为 Targets/Bonito/conf/ld.script 。

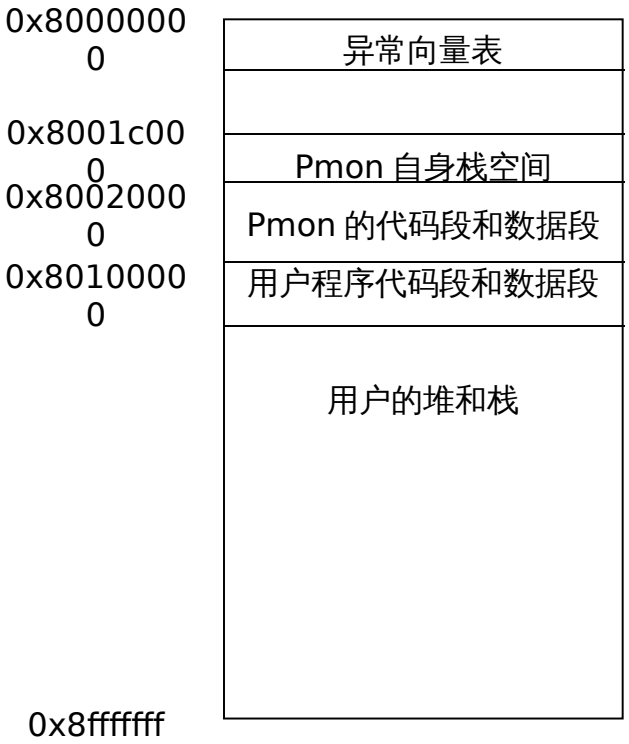
PMON2000 的框架



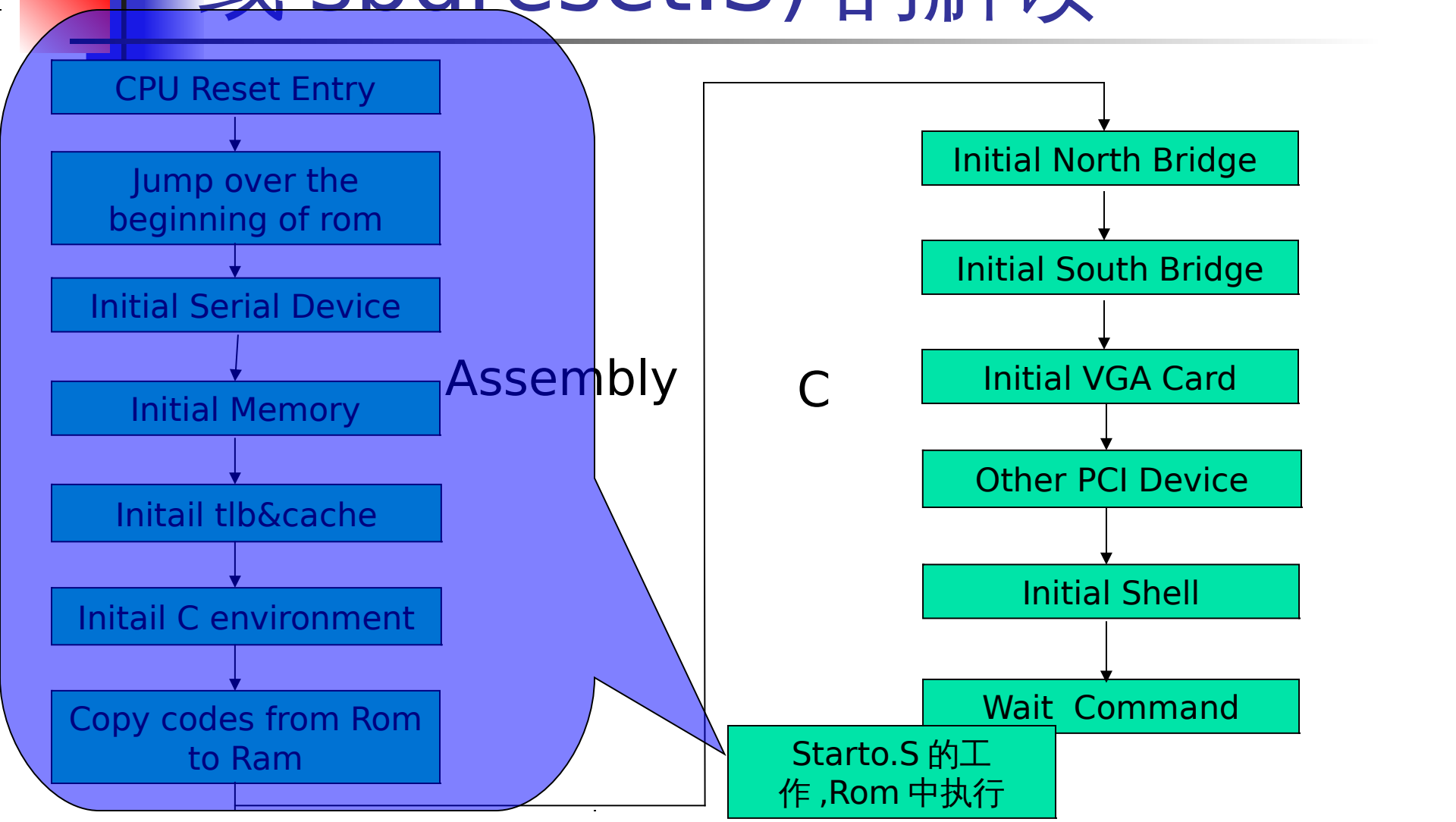
异常向量表

异常类型	进入点			
	SR(BEV)=0		SR(BEV)=1	
	程序地址	物理地址	程序地址	物理地址
Reset(启动) NMI(不可屏蔽中断)			0xBFC0 0000	0x1FC0 0000
TLB 替换 （ 32 位）	0x8000 0000	0x0	0xBFC0 0200	0x1FC0 0200
XTLB 替换 （ 64 位）	0x8000 0080	0x80	0xBFC0 0280	0x1FC0 0280
Cache 错	0xA000 0000	0x100	0xBFC0 0300	0x1FC0 0300
普通中断	0x8000 0180	0x180	0xBFC0 0380	0x1FC0 0380

Pmon 的空间分配


0x80000000
异常向量表
0x8001c000
Pmon 自身栈空间
0x80020000
Pmon 的代码段和数据段
0x80100000
用户程序代码段和数据段
用户的堆和栈
0x8fffffff

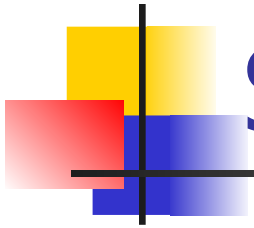
PMON 的汇编部分 (starto.S 或 sbdreset.S) 的解读



Start.S

- 1. 当整个板子起电后 ,CPU 将从 0xBFC00000 取指令开始执行 , 而 ROM 在系统中的地址就是从该地址开始的 , 所以其中的第一条指令就是整个 CPU 的第一个指令 , 在 MIPS 中 , 异常处理入口有两套 , 通过 CP0 的 STATUS 寄存器位 BEV 来决定 , 当 BEV=1 时 , 异常的入口地址为 0xBFC00000 开始的地址 , 而 BEV=0, 异常地址为 0x80000000 开始的地址 , 所以 PMON 程序段开始处是一些异常的调入口，需要跳过这段空间，程序通过一个跳转 bal 指令跳到后面 .

```
bal     locate
nop
```



Start.S

```
        bal         uncached
        nop
        bal         locate
        nop
uncached:
        or      ra, UNCACHED_MEMORY_ADDR
        j       ra
        nop
```

此处是可以从 cache 空间转换到 uncache 的空间 ,ra 中保留的是 bal locate 这条指令的地址 , 然后或上 UNCACHED_MEMORY_ADDR, 该地址就变成 uncache 的地址了 .

Start.S

- la s0, start
  subu s0, ra, s0
  and s0, 0xffff0000

这段代码是为了访问数据，因为这段汇编在 Rom 执行，而编译出来的数据段在 0x8002xxxx, 为了能够访问数据段的数据，需要进行一个地址的修正， s0 这是起到这种修正的目的。

Start.S

- 初始化 CPU 内的寄存器，清 TLB.
- 初始化一些北桥的基本配置，以确保 uart 能够正常工作 .
- 初始化 uart, 主要是设置波特率 .
- 初始化内存 (主要通过 I2C 协议从内存的 EEPROM 读取内存参数来进行设置).
- 初始化 cache.
- 拷贝 pmon 的代码到内存，然后通过

```
la      v0, initmips
jalr    v0
nop
```

从此代码便到内存中间去了，从这开始因为可以读写内存，所以有了栈，故可以用 C 的代码了，所以以后的程序便是 C 代码了 .

C 代码部分

- Ram 中运行，入口为 initmips

在文件 Targets/Boniton/Bonito/tgt_machdep.c 中

```
void
initmips(unsigned int memsz)
{
      ...
      tgt_cpufreq();
      cpuinfotab[0] = &DBGREG;
       dbginit(NULL);

      bcopy(MipsException, (char *)TLB_MISS_EXC_VEC, MipsExceptionEnd - MipsException);
       bcopy(MipsException, (char *)GEN_EXC_VEC, MipsExceptionEnd - MipsException);

       CPU_FlushCache();

       CPU_SetSR(0, SR_BOOT_EXC_VEC);
      main();
}
```

主要初始化在 dbginit 函数中执行。

dbginit

```
void
dbginit (char *adr)
{
    __init();       /* Do all constructor
    initialisation */
    envinit ();


    tgt_devinit();


#ifdef INET
    init_net (1);
#endif


#if NCMD_HIST > 0
    histinit ();
#endif
```

```
#if NMOD_SYMBOLS > 0
            syminit ();
#endif
 #ifdef DEMO
            demoinit ();
#endif


            initial_sr |= tgt_enable
(tgt_getmachtype ());


#ifdef SR_FR
       Status = initial_sr & ~SR_FR; /*
don't confuse naive clients */
#endif
            ioctl(STDIN, TCGETA,
&consterm);
...
...
...
} //gdbinit
```

dbginit

__init(); 初始化一些数据结构 .

Envinit (); 初始化环境变量 .

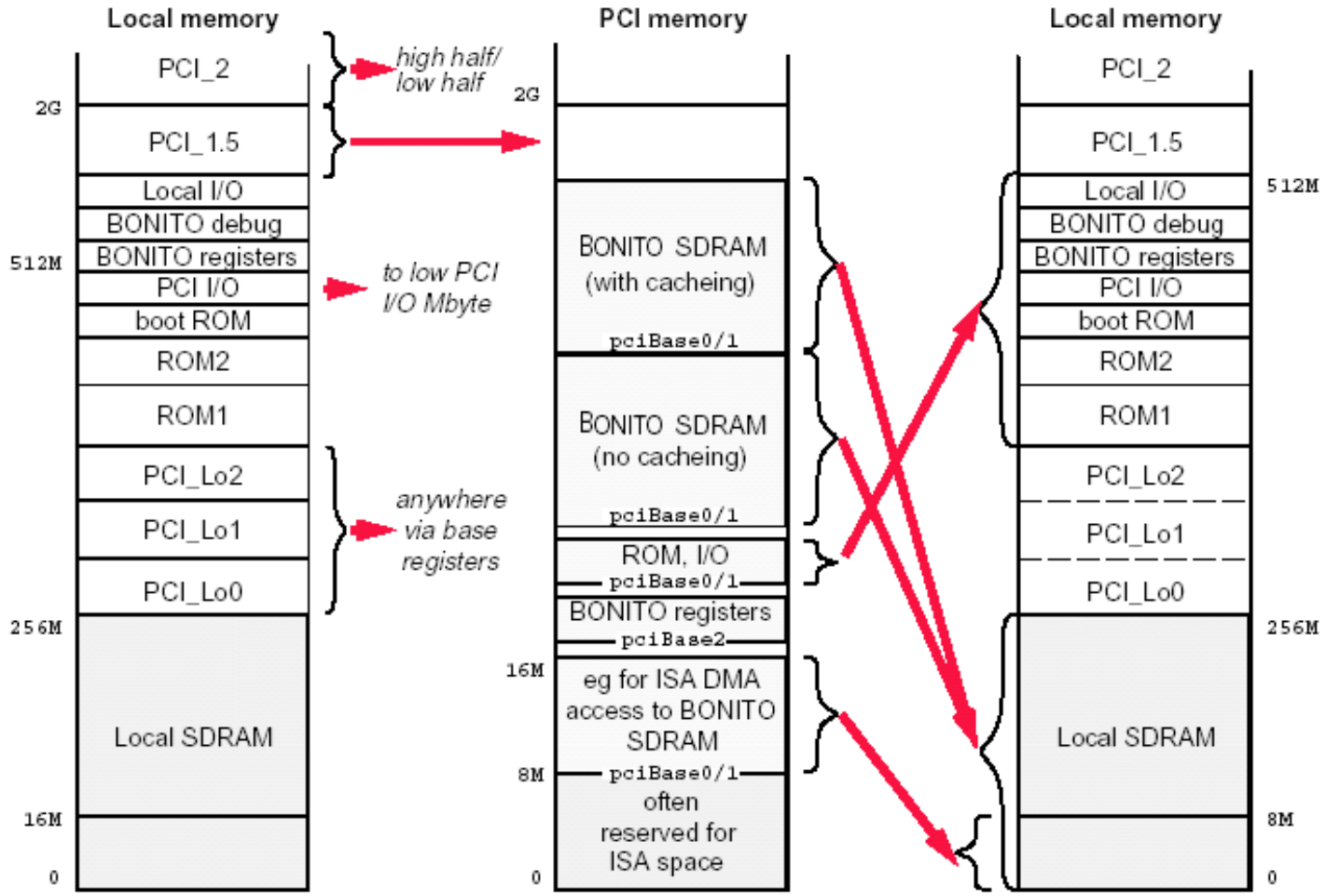
tgt_init(); 初始化与板级相关的过程，在我们系统中主要是初始化北桥和 PCI.

inet_init(); 初始化网络 .

Hisinit(); 初始化命令历史记录 .

Ioctl(STDIN,TCGETA,&consterm); 建立终端 .

Bonito 的空间分配



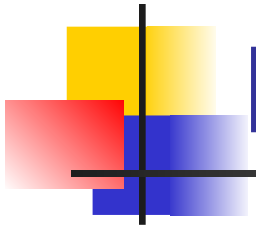
PCI 的空间分配

- tgt_devinit()→_pci_businit() →_pci_hwinit
- pci_hwinit() 为 Pmon 主要初始化 PCI 在北桥的窗口的函数 ，这个函数在 Target/Ev64240/pci/pci_machdep.c 中定义

```
pd = pmalloc(sizeof(struct pci_device));
  pb = pmalloc(sizeof(struct pci_bus));

  pd->pa.pa_flags = PCI_FLAGS_IO_ENABLED | PCI_FLAGS_MEM_ENABLED;
  pd->pa.pa_iot = pmalloc(sizeof(bus_space_tag_t));
  pd->pa.pa_iot->bus_reverse = 1;
  pd->pa.pa_iot->bus_base = PCI0_IO_SPACE_BASE - PCI0P_IO_SPACE_BASE;
  pd->pa.pa_memt = pmalloc(sizeof(bus_space_tag_t));
  pd->pa.pa_memt->bus_reverse = 1;

  pd->pa.pa_memt->bus_base = 0;
  pd->pa.pa_dmat = &bus_dmamap_tag;
  pd->bridge.secbus = pb;
  _pci_head = pd;
```

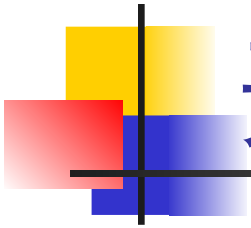


PCI 空间分配

```
pb->minpcimemaddr  = PCI0P_MEM_SPACE_BASE;
 pb->nextpcimemaddr = PCI0P_MEM_SPACE_BASE + PCI0_MEM_SPACE_SIZE;
 pb->minpciioaddr  = PCI0P_IO_SPACE_BASE;
 pb->nextpciioaddr = PCI0P_IO_SPACE_BASE + PCI0_IO_SPACE_SIZE;
 pb->pci_mem_base   = PCI0_MEM_SPACE_BASE;
 pb->pci_io_base    = PCI0_IO_SPACE_BASE;
```

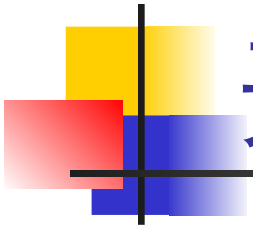
建立 PCI 的空间分配的数据结构 . 其中 pci_mem_base 为 Memory 空间的基地址 ,pci_io_base 为 IO 空间的基地址 .minipciioaddr 为 IO 空间的最小可以分配地址 ,minipcimemaddr 为 Memory 空间的最小可以分配地址 .nextpcimemaddr 为 PCI 的 Memory 空间的下一个分配地址 ,nextpciioaddr 为 PCI 的 IO 空间的下一个分配地址 , 在 pmon 的中地址分配是逆序分配的 .

```
pb->max_lat = 255;
 pb->fast_b2b = 1;
 pb->prefetch = 1;
 pb->bandwidth = 4000000;
 pb->ndev = 1;
 _pci_bushead = pb;
 _pci_bus[_max_pci_bus++] = pd;
```



开发时需要注意的问题

- Debug 的方法
  - 在串口设备没有初始化前，利用逻辑分析仪进行测试。该方法 debug 很艰难，所以应该尽早初始化串口。
  - 串口工作后，可以利用串口进行类似 printf 的 Debug 方法。
  - 初始化 BIOS 系统的 Shell 后，可以利用 BIOS 系统中的 Debug 系统进行测试。



开发时需要注意的问题

- 地址空间是否正确，北桥上的窗口分配是否正确。
- 中断问题。
- 充分利用 Pmon 所带的调试手段。 pmon>h 可以看到 pmon 提供的命令，通过这些命令来检查地址分配是否正确。