# Summary Questions

## 1 GMINT module

### 1） No memory found(double check):

**when check data, can not find the same address in the global memory.**

1351 ERROR : No memory found -> chk( axi\_slave) a=00026a1000 d=00f3e301 b=1001

……

原因是由于在instance GMINT.hdr时，没有区分Channel A 和Channel B,导致同样一笔write data，Channel A check后，Channel B 也同样check了，但Channel B上并没有write data，因此在memory上无法找到这笔data的address，从而导致了这个ERROR。

1330 top.axi\_slave\_CHA.chkdatapm chkdata : DRAMCA, a=0097d51000, d=462df78c, 0000

1330 ERROR : No memory found -> chk( DRAMCA) a=0117d51000 d=462df78c b=0000

[gfx\_dram\_hp\_flh\_mon\_CHA INFO:@ 1310050] address 0097d51000 hit nxm

1314 wrt( MXU) a=0097d51000 d=462df78c b=0000 <- 0097d51000

1317 chk( DVAD\_MXU\_A\_mon) a=0097d51000 d=462df78c b=0000 i= 0 <- 0097d51000

\*该笔数据由MXU->DRAMC的write cycle，地址为0097d51000。

\*该地址需要写一笔462df78c的数据，但在DRAMC上该笔数据对应的地址是0117d51000。

\*由此产生了ERROR，找到同样的data，但address却不是那个address。

>解决方案：只是monitor报的false ERROR, 因为SVAD2DVAD的mapping，会将地址自动减2G, 所以前后address可以对的上。产生这个的ERROR的根源在于make chklst的时候，没有define TAD\_FUNCTION\_OPEN，让环境以为TAD也就是DVAD没有打开，对应的monitor也没有吃进去。

### 2） No matched item:

**when check data, can find the same address in the global memory, but data or be not match.**

ERROR : no matched item -> chk(MXU) a=0010776940 d=1637e12c b=0000 who=CPUs in GMEM:

由于数据传输是MXU->DRAMC，所以这个ERROR是read channel产生的。首先找data的request地址，发现也是0010776940，之后在waveform中查找这个地址，发现一个ID（1 burst）所对应的RVALID没有连起来，出现了间断。由此造出根据RVALID和RLAST产生的RHEAD在一个ID期间出现两个，造成了data传输到MXU的错误。

3078 ERROR : no matched item -> chk( MXU) a=00264df740 d=fde2d2fc b=0000 who=MASTERs in GMEM:

3078 gmem[ DRAMCA] a=00264df740 d=a777424f b=0000 who= at 3056.290 ns

\*该笔数据是由DRAMC->MXU的read data back，地址为00264df740。

\*产生ERROR的原因是DRAMC返回read data应该为8QW，但前4QW由于dynamic起来较晚没有采到，造成了data的部分丢失。 a777424f才是00264df740地址对应的数据，但是由于前4QW丢失，所以后4QW对应的地址00264df740 所对应的数据递补上去，即fde2d2fc。

\*因此造成了data的匹配错误，找到同样的address，data却不是那个data。

>解决方案：model需要改下，DRAMC不应该收到request后就立即返回数据，应该间隔5T DCLK左右。

### 3） DPR

T-SEG : Only CPU access memory range; (Programmable)

Frame buffer : Only GFX can access memory range; (Programmable)

DPR : DMA (Memory) protect range, device can’t access; (Below 2G)

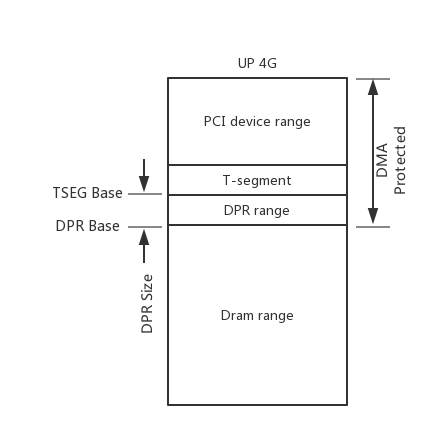
DMA : Direct memory access;

If the memory range is 0~2G, SVAD = DVAD;

If the memory range is above 4G, SVAD-2G=DVAD;

If the memory range is 2G~4G, the request will be abort.

NXM: SVAD->DVAD需要abort的开关



一般情况下，低于4G, SVAD = DVAD, 大于4G,才会SVAD-2G=DVAD.(假定不能不访问的区域为2G)

SVAD: system view address decoder;

DVAD: device view address decoder.

### 4） Downstream and Upstream

downstream: 即C2P, CPU发cycle给下面的device；

upstream: 即P2C，下面的device主动发cycle给CPU。

对于GMINT来讲，只有downstream, 不过master是MXU。因为slave是DDR，是不能发cycle的。有upstream的device比如PCIe，XHCI等。

snoop:经过cpu的cycle,需要cpu来参与，保持数据的一致性；

no-snoop:不经过cpu的cycle。

### 5） Over grasstime bug:

45078.5 ERROR : over grasstime -> a=003856cd18 d=56b403ad b=1101 who=AGPMAS at 934.870000ns （gmint\_basic\_max.src）

原因是checklist的for循环没有退出导致的。ex:

reg [2:0] iii;

for(iii=0;iii<8;iii++)

这里错误定义了iii，3bit的width不会超过7，由此造成恶性循环。

解决方案：将iii定义为interge类似，即integer iii; 或拓展1bit，即reg [3:0] iii;

推荐第一个方案。

### 6） Mismatch cycles

[gfx\_dram\_hp\_flh\_mon\_CHB ERROR:@ 949530] mismatch cycles

[gfx\_dram\_hp\_flh\_mon\_CHB ERROR:@ 949530] expected: addr 016ae7a100, len e

[gfx\_dram\_hp\_flh\_mon\_CHB ERROR:@ 949530] real: addr 0055296e60, len 1

monitor的原因，找相关owner解决。

### 7） Address Range transfer

1B-> 2^0 -> ‘b1-> ‘h1

1K-> 2^10-> ‘b100\_0000\_0000-> ‘h400

1M-> 2^20-> ‘b10000\_0000\_0000\_0000\_00000-> ‘h10\_0000

1G-> 2^30-> ‘b100\_0000\_0000\_0000\_0000\_00000\_0000\_0000-> ‘h4000\_0000

bank0~7 top address:

bank0: ‘h1000\_0000-> 2^28 -> 250M

bank1: ‘h2000\_0000-> 2^29-> 500M

bank2: ‘h3000\_0000-> 2^29+2^28-> 750M

bank3: ‘h4000\_0000-> 2^30-> 1G

bank4: ‘h5000\_0000-> 2^30+2^28-> 1.25G

bank5: ‘h6000\_0000-> 2^30+2^29-> 1.5G

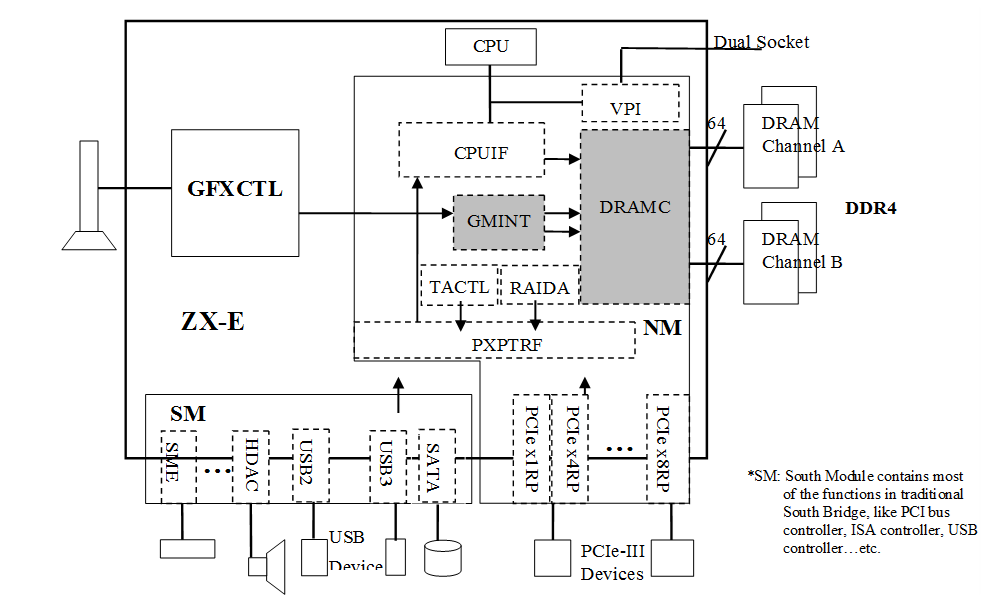
bank6: ‘h7000\_0000-> 2^30+2^29+2^28-> 1.75G

bank7: ‘h8000\_0000-> 2^31-> 2G



### 8）Structure in NB





详见《SOC记》

### 9) IRS

`PROCESSOR.cfgw('h0000\_01e8,4'b1101,(`BIT32HIGH<<15) | (({`MODUTOP.RSNPOERRSMIEN,`MODUTOP.RSNPOERRNMIEN,`MODUTOP.RSNPOERRSCIEN}|32'b0)<<8));

01e8:表示D0F1, 1101表示BE,低有效，这里表示e9处的register有效。后面为配置的内容，即MODUTOP处的register值。

SNPO Address Error Status CHA

Set when an address of SVA not in FBACS and RGMISNPOEN is enabled.

0: Not detect

1: Detectet

It can trigger SMI/NMI/SCI event when enable bit is set.

@((control = RSNPOAERRSTATUS\_I))

((For Internal Reference: @((#USER=GMINT))

该寄存器为1，表示检测SVA的地址是否在framebuffer中，如果不在则报error。此时需要设置SMI/NMI/SCI event.

## 2 Simulation Environment

### 1） make GMINT checklist时，出现以下问题：

Dumping VCS Annotated Stack:

#0  0x00007fffefaae945 in pthread\_cond\_wait@@GLIBC\_2.3.2 () from /lib64/libpthread.so.0

#1  0x00007ffff05362b3 in SNPSle\_95c894712e6f9118bc51d0bbc485432a () from /cad/SYNOPSYS/vcs-mx\_vL-2016.06-SP2/linux64/lib/libuclinative.so

#2  0x00007ffff053669b in SNPSle\_de8e8fbcd7ecf84c9ae14573df0a2ec6 () from /cad/SYNOPSYS/vcs-mx\_vL-2016.06-SP2/linux64/lib/libuclinative.so

原因是由于之前跑simulation的时候，有些临时文件影响了VCS的进程。

该问题的解决方法如下：

rm -r work\_\*

rm -r simv\*

rm -r csrc/

注意：如果使用rm -i filename:表示删除文件前先确认。

### 2） 版本控制（mercurial）：

**mercurial版本控制工具：**

hg st: 用来查看文件是否改动；

hg lg: 查看版本，使用q退出；

hg pull: 将central的高版本拉到local中；

hg up: 将高版本和local的低版本（与central对应的低版本相比修改过）进行merge。正常来讲，如果merge失败后hg工具会禁止commit，需人工merge并hg resolve –mark后才可继续；

hg ci –m “\*\*\*\*”：用来commit提交产生的最高版本；

hg rollback:用来撤销提交的最高版本； （commit后，push前后都行）

hg push:将最高版本push到central中，此时将不可更改；

hg revert aa: hg revert aa用来恢复改动的文件aa，和central的文件一样；（commit 前）

hg id –n:查看当前版本号；（带+号表示该版本有改动）

hg -q tip：显示当前版本号的详细信息；

hg export tip：显示当前版本号的改动痕迹；（前后两个版本比较）

diff -r a5678e77c3aa -r 2377f00e6753 DMAXI.hdr:比较任意两个版本的改动。

（或hg diff -r 82:83 DMAXI.hdr，本质上也是执行上面两个命令）

hg remove+文件名:删除central中的文件；

hg revert --all: 恢复所有改动的文件，包括remove的；

hg resolve -m:如果merge发生conflict，需要手动修改文件，并且用该命令解决，然后commit；

hg tag v1.0:对于重要的版本，用该命令作为tag，“hg up v1.0”可以用来恢复该版本；

hg diff + 文件名：显示改动的痕迹；发生在pull up之前。

hg init: 建立central环境；

hg clone: copy central “ /cpuwrk/dip/sim/repo/GMINT ” 到“/cpuwrk/NB\_T3/users/nb/ericwang/” 路径下generate GMINT\_central(可以换成new name)，然后在任一路径下“hg clone /cpuwrk/NB\_T3/users/nb/ericwang/GMINT\_central/”，则该路径下的GMINT\_central的central就是“/cpuwrk/NB\_T3/users/nb/ericwang/GMINT\_central/”；

hg path:显示所建立环境direct的central；

hg add: 增加文件到central；

其他版本控制常用命令：

cp instance.v instance.v.版本号：copy文件进行备份

hg up –r +版本号 ：将某版本的文件与现在的文件进行更新 （push 后）

g –d instance.v instance.v.版本号：比较两个文件

目前环境的建立方式：

1 有一个真正的central放在repo下: /cpuwrk/dip/sim/repo/GMINT；

2 对repo下的central 进行“hg clone”，产生：/cpuwrk/dip/sim/GMINT；

3 使用脚本cp /cpuwrk/dip/sim/GMINT这个目录，生成我们需要的环境，但生产的环境的hg path仍然都是/cpuwrk/dip/sim/repo/GMINT。

注意;central中的文件不能动，只能在新拿的环境中增减或修改，再push。

使用hg update命令来更新central中的文件。

**Git版本控制工具：**

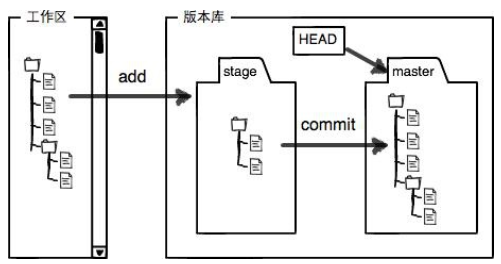
1 git init: 在任一目录下type该命令，表示建立git版本控制系统；

2 文件的修改、增添和提交的组合命令为：git add + 文件名；git commit –m “xxxxx”;

3 git status:显示目前的状态；

4 git diff + file name: 显示修改后的痕迹。当在工作区修改文件后，add之前可以用该命令显示做了哪些改动；

5 git 版本控制分为两大部分：工作区和版本区，工作区到stage（暂存区）的命令是add， stage到master的命令是commit；



6 版本的退回，时光车:

git reset –-hard HEAD^: 退回到上1个版本；

git reset –-hard HEAD^^: 退回到上2个版本；

git reset –-hard HEAD~10: 退回到上10个版本；

git reset –-hard+版本号前几位数字: 退回到某一个版本；

7 显示log的方法：

git log:显示从最近到最远的提交记录；

git log --graph:显示分支合并图；

git log -1:直线式最后一次提交的信息；

git reflog:记录每次命令，可以得到每次命令的版本号，可用来恢复版本；

git log --pretty=oneline:显示一行

8 撤销修改的三种方式方法：

git checkout -- readme.txt: 把工作区的修改撤销，即让文件回到最近一次git commit或git add时的状态，这里好像也可以使用git checkout readme.txt；（add 前）

git reset HEAD readme.txt: 把stage区的修改改回到工作区时的状态；（add后commit前）

git reset --hard HEAD^: 退回上一个版本。（commit后）

9 删除文件后提交的组合命令：

git rm+filename； git commit -m”xxxx”；

或：rm+filename; git add --all ; git commit -m”xxxx”;

10 分支的建立：

git branch dev :建立分支dev，和默认的主支master相对；

git checkout dev: 切换到分支dev；（dev的name可变）

（以上两个命令可合并为：git checkout –b dev）

git branch: 查看当前分支；

git checkout master:切换到主支上；(dev上的改动必须commit才可以切回来)

git merge dev: 将分支dev上的改动合并到主支master上；

git merge --no-ff -m”注释” dev: merge dev分支的时候，禁用fast-forward，保留分支信息。

git branch –d dev:删除分支dev；（-d改为大写表示强制删除）

git config --global alias.st status: 全局设置简称，可以用st来代替staus；u

### 3） dump.tcl和dump\_vcs.tcl

注意区分两个dump文件，如果采用的是vcs仿真，需要看的是dump\_vcs.tcl; 如果采用的是irun仿真，需要看的是dump.tcl。

另外，看makefile时，需要看include的文件，防止需要替换文件时，没有替换干净。

### 4） VCS/IRUN and FSDB file

这两个都是simulation tool，各有自己的特点。

目前发现的情报是：IRUN可以报出重复定义的ERROR，VCS不可以；VCS的simulation要比IRUN快；

两者的dump.tcl文件格式也有区别（注意大括号的位置及$符号）。

比如VCS的如下：

call {$fsdbDumpfile ("twave\_basic.fsdb")};

#run 35000ns

#run 0

#save snap35

IRUN的如下：

call fsdbDumpfile {"twave\_make.fsdb"};

#run 35000ns

#run -clean

#save -simulation snap35

FSDB (Fast Signal DataBase)是通过verilog的PLI接口来实现的，例如$fsdbDumpfile, $fsdbDumpvars等，它们各自代表的含义如下：（黄色标记表示可以独自产生波形文件）

fsdbDumpfile - 指定FSDB文件名

fsdbDumpvars - Dump指定的变量

*call {$fsdbDumpvars (0, "top", "+skip\_cell\_instance=1") };*

*0代表level，top代表hierarchy，最后一个代表变量(optional)。*

fsdbSwitchDumpFile - 将dumping切换到另一个FSDB文件

fsdbAutoSwitchDumpfile - 限制文件大小并在数据量过大时自动创建新的FSDB文件

*call {$fsdbAutoSwitchDumpfile ( 5000, "basic1.fsdb" ,100) };*

*5000代表file size(单位：M), 中间为file name, 最后为file number.*

*ps:后面也可以加一个log file name, 比如：*

*call {$fsdbAutoSwitchDumpfile ( 5000, "basic1.fsdb" ,100, “fsdb\_dump.log”) };*

$fsdbDumpon - 打开 FSDB dumping

*call {$fsdbDumpon}; call fsdbDumpon(IRUN);*

$fsdbDumpoff - 关闭 FSDB dumping

*call {$fsdbDumpoff}; call fsdbDumpoff(IRUN);*

fsdbDumpMem - Dump 指定的memory的内容

fsdbDumpflush - Force to Dump Result to FSDB file

fsdbDumpSingle - Dump指定的信号

fsdbDumpvariable - Dump指定的VHDL变量

fsdbDumplimit - 限制FSDB文件size

### 5） get env

[‎2018/‎6/‎13 13:41] Yajun Zhang(BJ-RD):

/cpuwrk/chx002/sim/get\_chx002\_vcs (chiplevel env)

/cpuwrk/NB\_T3/chx002\_sim\_bak/get\_chx002 这个脚本拿的环境是没有做过安全审查的

/cpuwrk/chx002/sim/get\_chx002 这个脚本拿的环境是在 7.15 基础上做过安全审查

### 6） 软链接

建立的形式：ln -s 源文件 目标文件

不加s表示硬链接，区别在于删除源文件后，硬链接的文件会保留，软连接的会变成空文档。

只有删除源文件和目标文件，硬链接才真正地被删除。

软链接的形式如下所示，文件后带有@：

GMINT.v@ GMINTA.hdr@ GMINTB.hdr@ mixcnd001.cfg@

type 命令lt （alias l –tr）, 或者ls -li用来display软链接的文件，

其中ls -li不显示隐藏文件，如下所示：

GMINT.v -> ../src/arc/GMINT.v

GMINTA.hdr -> ../src/arc/GMINTA.hdr

GMINTB.hdr -> ../src/arc/GMINTB.hdr

mixcnd001.cfg -> ../src/arc/mixcnd001.cfg

软连接的文件相当于一份文件，改任意一个都会在另一个文件中同样改动。类似于量子纠缠效应。

### 7） Verdi tip

Verdi 界面分成两大部分：nTrace和nWave，前者是RTL，后者是waveform.

nTrace: m键用来显示macro（define的宏变量）；

x键用来显示active annotation,用来和nWave界面的波形进行对照；

e键用来打开编辑，现改文件。

nWave: H键用来显示signal的hierarchy；

m键用了移动选中的signal到选的的位置（黄线标记出）；

shift+s用来保存waveform 拉好的信号，

r键用来加载保存好的waveform信号；

o键用来加载dump好的waveform文件，

shift+l用来重新加载dump的waveform文件；

c键用来选择波形的颜色；

### 8） Linux operate

#### 1、find

find –name +文件名，表示寻找本层或下层文件，注意不能搜寻上层或同层文件。

find -iname filename: 同上，不区分大小写；

find -empty: 查找空文件。

#### 2、echo

echo $USER 显示用户名；

echo $PATH 所有显示路径；（echo $PATH | sed 's/\:/\n/g'用来更方便地观察）

#### 3、which

which verdi :显示verdi的版本号；也可以是其他tool，比如vcs。

which ggmint:显示alias ggmint的原始命令。

#### 4、wc

wc -l + 文件名：显示文件有多少行；

#### 5、cat

cat file1 file2 > file3:两个文件合并成file3，如果file3中有内容则覆盖；

cat file1 file2 >>file3:两个文件附加到file3中；

参数：

-n 或 –number 由 1 开始对所有输出的行数编号

-b 或 –number-nonblank 和 -n 相似，只不过对于空白行不编号

-s 或 –squeeze-blank 当遇到有连续两行以上的空白行，就代换为一行的空白行

-v 或 –show-nonprinting

nl + 文件名：和cat+文件名一样，只不过会对其编号，和cat -b的功能一样。

#### 6、du -sh

du -sh + 文件名：显示文件大小

du -h filename: 方便阅读的格式显示test目录所占空间情况

#### 7、sed 的使用

sed [-hnV][-e<script>][-f<script文件>][文本文件]

参数说明：

-e<script>或--expression=<script> 以选项中指定的script来处理输入的文本文件。

-f<script文件>或--file=<script文件> 以选项中指定的script文件来处理输入的文本文件。

-h或--help 显示帮助。

-n或--quiet或--silent 仅显示script处理后的结果。

-V或--version 显示版本信息

动作说明：

a ：新增， a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～

c ：取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！

d ：删除，因为是删除啊，所以 d 后面通常不接任何咚咚；

i ：插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；

p ：打印，亦即将某个选择的数据印出。通常 p 会与参数 sed -n 一起运行～

s ：取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配正规表示法！例如 1,20s/old/new/g 就是啦！

比如：

sed -n '/ZX/p' gmi\_pftest\_01.src ： 显示带有ZX字符的行；

sed -e 4a\newLine testfile ：在testfile的第四行后新增一行“newLine testfile”；

（或者sed -e “4a newline” testfile，又或者sed -e ‘4a newline’ testfile）

sed -i 4i\ newLine testfile ：在testfile的第四行前新增一行“newLine testfile”；

（**注意，sed 的『 -i 』选项可以直接修改文件内容**）

nl perl.v | sed -e '4a wang' ：增添显示行号功能，新增的‘wang’行不算进行数；

sed -i '$a # This is a test' aa.txt： 由于 $ 代表的是最后一行，而 a 的动作是新增，因此该文件最后新增『# This is a test』！

sed '2,5c No 2-5 number' filename： 2-5行，被“No 2-5 number”取代；

sed -n '5,7p' filename：打印5-7行；

sed -n ‘/ZX/p’ filename:搜索并打印带有“ZX”字符的行数；

sed -e ‘/ZX/d’ filename:搜索并删除带有“ZX”字符的行数；

(-e如果替换成-i，可在原文本上操作)

sed -i 's/要被取代的字串/新的字串/g' filename: 在原文本中替换字符；

sed -e '3,$d' -e 's/bash/blueshell/：一条sed命令，删除/etc/passwd第三行到末尾的数据，并把bash替换为blueshell，e表示多点编辑，第一个编辑命令删除/etc/passwd第三行到末尾的数据，第二条命令搜索bash替换为blueshell。

 “//  87463467e\* refs/head/master” ”//”

      推荐快速删除方法如下： sed  “s#^\/\/ .\*refs\/heads\/master#\/\/g” \*.v –i –r

#### 8、grep的用法：

grep -r “example” \*: 在一个文件夹中递归查询包含指定字符串的文件；

grep -i “example” filename: 查找字符时不区分大小写；

grep -A 3 -i “example” filename:输出成功匹配的行，以及该行之后的3行。

统计文件数：ll | grep "^-" -c 或 ls -l | grep "^-" -c

统计目录数：ll | grep "^d" -c

Linux常见的文件类型有：普通文件、目录文件、字符设备文件和块设备文件、符号链接文件等，其中：

1 普通文件的文件权限第一个字符为“-”

2 目录文件的文件权限第一个字符为“d”

3 字符设备文件的文件权限第一个字符为“c”；

4 块设备文件的文件权限第一个字符为“b”

5 符号链接文件的文件权限第一个字符为“s”

#### 9、拆分文件

1. 以行数拆分 -l 参数：

split –l 50 原始文件 拆分后文件名前缀

例：以50行对文件进行拆分

split -l 50 big.txt small\_

拆分后会生成 small\_aa small\_ab small\_ac ...

2. 以大小拆分 -b 参数：

例：每个文件1M

split –b 1024 big.txt small\_

#### 10、terminal下打开pdf文件

evince [filename]

linux下打开图片文件的命令 eog全称：eye of gmone，是linux下内置的图片查看器。

#### 11、cp

比如cp file1~4到某一路径下，方式如下：

cp /home/usr/dir/{file1,file2,file3,file4} /home/usr/destination/

需要注意的是这几个文件之间不要有空格

如果具有共同前缀，可以采用以下方式：

cp /home/usr/dir/file{1..4} ./

cp -p file1 file2: 拷贝文件1到文件2，并保持文件的权限、属主和时间轴；

cp -i file1 file2: 拷贝file1到file2，如果file2存在会提示是否覆盖.

#### 12、df查看磁盘空间

df + 参数 + 目录或文件名， 参数如下：

-a ：列出所有的文件系统，包括系统特有的 /proc 等文件系统；

-k ：以 KBytes 的容量显示各文件系统；

-m ：以 MBytes 的容量显示各文件系统；

-h ：以人们较易阅读的 GBytes, MBytes, KBytes 等格式自行显示；

-H ：以 M=1000K 取代 M=1024K 的进位方式；

-T ：显示文件系统类型, 连同该 partition 的 filesystem 名称 (例如 ext3) 也列出；

-i ：不用硬盘容量，而以 inode 的数量来显示



#### 13、chmod

在Linux中我们可以使用ll或者ls –l命令来显示一个文件的属性以及文件所属的用户和组，如：

**-rw-r--r--** 1 ericwang CPU-PD2BJ-Logic24 34577781 Sep 28 19:38 irun.log

**lrwxrwxrwx** 1 ericwang CPU-PD2BJ-Logic24 15 Sep 28 19:25 makefile -> ../env/makefile

**drwxr-xr-x** 2 ericwang CPU-PD2BJ-Logic24 155 Sep 28 19:25 rc

在Linux中第一个字符代表这个文件是目录、文件或链接文件等等。

当为[ d ]则是目录

当为[ - ]则是文件；

若是[ l ]则表示为链接文档(link file)；

若是[ b ]则表示为装置文件里面的可供储存的接口设备(可随机存取装置)；

若是[ c ]则表示为装置文件里面的串行端口设备，例如键盘、鼠标(一次性读取装置)。

接下来的字符中，以三个为一组，且均为『rwx』 的三个参数的组合。其中，[ r ]代表可读(read)、[ w ]代表可写(write)、[ x ]代表可执行(execute)。 要注意的是，这三个权限的位置不会改变，如果没有权限，就会出现减号[ - ]而已。

三组分别代表：属主（该文件的所有者），属组（所有者的同组用户）以及其他用户拥有的权限，用字母分别表示为：owner/group/others。

所以如果我们需要将文件权限设置为 -rwxr-xr-- ，可以使用

chmod u=rwx,g=rx,o=r 文件名

来设定。但使用 “chmod 754 文件名” 更为方便些。

chmod ug+rwx filename: 给指定文件的属主和属组所有权限(包括读、写、执行)

chmod g-rwx filename: 删除指定文件的属组的所有权限

chmod -R ug+rwx \*: 修改目录的权限，以及递归修改目录下面所有文件和子目录的权限

新建目录的权限默认为：drwxr-xr-x，即755，因为r的权重为4，w的为2，x的为1，分别代表可读，可写，可执行；

新建文件的权限默认为：-rwxr-xr--，即754.

#### 14、awk

删除重复行：awk '!($0 in array) { array[$0]; print}' + filename

打印文件中指定部分：awk '{print $2,$5;}' filename

#### 15、sort

sort + filename: 以升序对文件内容进行排序；

sort -r + filename: 以降序对文件内容进行排序；

#### 16、ls的解读

ls -lh:以易读的方式显示文件的大小，比如以MB,GB为单位：

drwxr-xr-x 2 ericwang CPU-PD2BJ-Logic24 4.0K Sep 29 15:29 arc

ls -ltr :以最后修改时间升序列出文件；（或者ll -tr）

ls -lt: 以最后修改时间升序列出文件；（或者ll -t）

ls -F：在文件名后显示文件类型。比如gmint sim目录下的4中类型：

remake@（软连接）work\_gmint/（文件夹）makefile\*（可执行文件）irun.log（普通文件）

ls -d \*.vdb: -d选项用来针对目录的现实，如果不带-d，则用来显示文件list

ls -1：只是显示文件名，没有权限等信息

#### 17、ps命令

用于显示正在运行中的进程信息，比如：

ps -ef | grep vim: 查看带有vim字样的进程；

ps -ef | more :查看当前正在运行的所有进程；

ps -efH | more: 以树状结构显示当前正在运行的进程，H选项表示显示进程的层次结构.

该命令经常与kill一起使用，ps -ef查找到某个进程的进程号后，kill+进程号来终结该进程。

#### 18、 free命令

用来显示系统当前内存的使用情况，包括已用内存、可用内存和交换内存的情况，默认情况下free会以字节为单位输出内存的使用量

如果你想以其他单位输出内存的使用量，需要加一个选项，-g为GB，-m为MB，-k为KB，-b为字节；

如果你想查看所有内存的汇总，请使用-t选项，使用这个选项会在输出中加一个汇总行。

cpu0751:/c0751/ericwang/CHX002\_GMINT\_0929150943/sim\_env/sim#[862]free -t -g

total used free shared buff/cache available

Mem: 31 2 18 0 10 28

Swap: 124 0 124

Total: 156 2 143

#### 19、top命令

top命令会显示当前系统中占用资源最多的一些进程（默认以CPU占用率排序）如果你想改变排序方式。

如果只想显示某个特定用户的进程，可以使用-u选项：top -u ericwang

#### 20、passwd:

passwd用于在命令行修改密码，使用这个命令会要求你先输入旧密码，然后输入新密码。

#### 21、mkdir

使用-p选项可以创建一个路径上所有不存在的目录,比如：mkdir -p aa/bb/cc

#### 22、uname

uname可以显示一些重要的系统信息，例如内核名称、主机名、内核版本号、处理器类型之类的信息,比如:

cpu0751:/c0751/ericwang/CHX002\_GMINT\_0929150943/sim\_env/sim#[898]uname -a

Linux cpu0751 3.10.0-693.21.1.el7.x86\_64 #1 SMP Wed Mar 7 19:03:37 UTC 2018 x86\_64 x86\_64 x86\_64 GNU/Linux

#### 23、whereis and whatis

当你不知道某个命令的位置时可以使用whereis命令，下面使用whereis查找grep的位置:

cpu0751:/c0751/ericwang/CHX002\_GMINT\_0929150943/sim\_env/sim#[906]whereis grep

grep: /usr/bin/grep /usr/share/man/man1/grep.1.gz /usr/share/man/man1p/grep.1p.gz

whatis显示某个命令的描述信息:

cpu0751:/c0751/ericwang/CHX002\_GMINT\_0929150943/sim\_env/sim#[907]whatis grep

grep (1) - print lines matching a pattern

grep (1p) - search a file for a pattern

#### 24、locate

locate命名可以显示某个指定文件（或一组文件）的路径，它会使用由updatedb创建的数据库.

locate+字符：用来显示所有带有该字符的路径，有点类似grep，但比grep更强大，不需要特定的路径就可以使用。

#### 25、zip打包解包

比如将 test 目录打包成一个文件，-r 表示递归打包包含子目录的全部内容，-q 表示安静模式，-o 表示输出文件，其后紧跟打包输出文件名：

zip -r -q -o test.zip /home/test

使用 du 命令查看打包后文件的大小：

du -h test.zip

使用 file 命令查看文件大小和类型：

file test.zip

1表示最快压缩但体积大，9表示体积最小但耗时最久，-x 排除上一次我们创建的zip文件，路径必需为绝对路径：

zip -r -9 -q -o test\_9.zip /home/test -x ~/\*.zip

使用 -e 参数可以创建加密压缩包：

zip -r -e -o test.zip /home/test

之后会要求输出密码及确认密码，解压时输出正确的密码方可解码

将 test.zip 解压到当前目录

unzip test.zip

使用安静模式，将文件解压到指定目录：

unzip -q test.zip -d ziptest

不想解压，只想查看压缩包的内容可以使用 -l 参数：

unzip -l test.zip

Linux 上面默认使用的是 UTF-8 编码,防止解压后出现中文乱码，要用参数 -O：

unzip -O GBK 中文压缩文件.zip

### 9） VIM operate

1 分屏：“ :sp + 文件名”，在已经打开的文件下，再次打开一个文件，两个文件呈上下结构进行对照；

2 打开路径文件： “gf”，如果一个文件中包含有另一个文件的路径，gf可以在原文件的基础上打开，进行覆盖。“br”为退出。如果一个terminal 下有多个table，“br”可能不会退到自己原来的文件上，此时可以通过“：buffers”查看文件号，然后”：b\*”返回;

3 vi +多个文件名 –p: 可以一次性打开所有的文件，每一个文件都有一个tabe.

4 vim的设置在家目录下的.vimrc文件中，比如：

set relativenumber： 设置动态行号，以光标处为基准；

set norelativenumber: 关闭动态行号

map <c-s> :source ~/.vimrc<Enter>：ctrl+c键表示刷新vim的配置；

map <F2> :setf verilog<Enter> ：把F2键设置成verilog的快捷键；

map <F3> <ESC>:w!<CR> ：把F3键设置为快速保存；

5 高效率完成100个输出信号的定义:

如果借助Vim编辑器的命令功能的话，只需要写下:

for ($i=0;$i<100;$i++) { print("output reg[$i:0] reg\_$i\n");}，

然后在同一行按shift+v，输入:!perl回车.

就能看到前面输入的那些代码被替换成了你本来想输入的100行内容。

6 其他技巧：

> 你是否曾经忘记在每行代码末尾加上”,”或”;”符号，编译失败后才发现需要逐行补上？如果使用Vim编辑器的话，只需要用shift+v选中需要修改的行，按下:’<,'>s%$%;%g就可以了，熟练之后整个过程不超过5秒钟。

> 你是否曾经在代码写完之后被老大臭骂一顿原因是你没有把所有的reg和wire定义都放到文件的统一位置（如第38行）？如果使用Vim编辑器的话，只需要使用:g%^\s\*reg\s\*%m 38加上:g%^\s\*wire\s\*%m 38就可以了。

> 你是否曾经被要求删除某个文件中所有的注释？只需要:%s%//.\*$%%g就可以了。

批量替换(from rena)：

RDSIADVOS0\_A[7:0]\_RK[3:0] = 1'b1;

RDSIADVOS0\_B[7:0]\_RK[3:0] = 1'b1;

的值为0：

:%s/RDSIADVOS0\_\(\w\d\)\_RK\(\d\) = 1'b1;/RDSIADVOS0\_\1\_RK\2 = 1'b0;/g

7 快捷键的使用：

I、A：移动到行首或行末的第一个字符处，并进入插入模式。

**;**：重复上一次的f查找操作

**.**：重复上一次的修改操作，跟;可以说是好基友，经常用来实现一些简单的重复操作，比录制宏要简单很多。经常有意识的使用这种简单重复，是vimer编辑思路进阶的分水岭。

%：移动到与当前括号匹配的括号处。

gj、gk：有时可视区域不够宽，而一行的字符有很多，导致了wrap。

那么通过j、k是无法直观的定位到同一物理行，却不在同一个可视行里的位置，

此时需要gj和gk。

8 编辑模式中，ctrl+p 可以选择已经出现的字符，防止不一致的type.

9 Vim打开多个tab窗口情况的切换：

:tabnew [++opt选项] ［＋cmd］ 文件            建立对指定文件新的tab

（:tabe + filename,也可以打开文件或者创造新文件）

:tabc       关闭当前的tab

:tabo       关闭所有其他的tab

:tabs       查看所有打开的tab

:tabp      前一个 (gT)

:tabn      后一个 (gt)

标准模式下：

gt , gT 可以直接在tab之间切换。

更多可以查看帮助 :help table ， help -p

vim——打开多个文件、同时显示多个文件、在文件之间切换

打开多个文件：

1.vim还没有启动的时候：

在终端里输入

vim file1 file2 ... filen便可以打开所有想要打开的文件

2.vim已经启动

输入

:open file 可以再打开一个文件，并且此时vim里会显示出file文件的内容。

同时显示多个文件：

:split （上下分屏显示）或：Sex

:vsplit （左右分屏显示）或：Vex

在terminal界面，使用：

vi file1 file2 -o:上下分屏显示

vi file1 file2 -O:左右分屏显示

在文件之间切换：

1.文件间切换

Ctrl+6—下一个文件

:bn—下一个文件

:bp—上一个文件

对于用(v)split在多个窗格中打开的文件，这种方法只会在当前窗格中切换不同的文件。

2.在窗格间切换的方法

Ctrl+w+方向键——切换到前／下／上／后一个窗格

Ctrl+w+h/j/k/l ——同上

Ctrl+ww——依次向后切换到下一个窗格中

10 vim折叠：

vim里的折叠命令都是以z开头，共6中折叠方式：

manual, indent, expr, syntax, diff, marker;

其中manual是最基本的，（但本.vimrc定义的是set foldmethod=marker）

首先，确认折叠功能开启：set foldenable;

创建折叠：zf结合光标移动命令（将光标放在函数的开括号上，用zf%可方便将函数折叠起来），折叠发生后产生带有注释的三个大括号；

切换折叠态：za， 或者打开用“大空格键”/zv，关闭用zc/zm；

删除折叠：zd；

11 标记

使用标记可以快速移动。到达标记后，可以用Ctrl+o返回原来的位置。 Ctrl+o和Ctrl+i 很像浏览器上的后退和前进。

m{a-z}: 标记光标所在位置，局部标记，只用于当前文件。

m{A-Z}: 标记光标所在位置，全局标记。标记之后，退出Vim， 重新启动，标记仍然有效。

`{a-z}: 移动到标记位置。

'{a-z}: 移动到标记行的行首。

`{0-9}：回到上[2-10]次关闭vim时最后离开的位置。

``: 移动到上次编辑的位置。''也可以，不过``精确到列，而''精确到行 。如果想跳转到更老的位置，可以按C-o，跳转到更新的位置用C-i。

`": 移动到上次离开的地方。

`.: 移动到最后改动的地方。

:marks 显示所有标记。

:delmarks a b -- 删除标记a和b。

:delmarks a-c -- 删除标记a、b和c。

:delmarks a c-f -- 删除标记a、c、d、e、f。

:delmarks! -- 删除当前缓冲区的所有标记。

:help mark-motions 查看更多关于mark的知识。

12 搜索多个关键词

/GCLK\|SCANEN\|…:vim中搜索多个key words;

search key words, 忽略大小写的方法：

方法1： 进入命令模式，输入命令“/string \c ”，将会查找string字符串。

如 :/write\c

方法2：命令模式下，输入命令 set ignorecase（或set noic），在没关闭该文件前提下，字符串的查找都将不区分大小写。

如 ：set ignorecase -> /write

13 搜索并替换一行中的第一个, 在命令模式下,使用:s/\*\*\*\*/\*\*\*\*/即可，或后面加个#.与普通替换s/\*\*\*/\*\*\*/g少一个g。

### 10） simulation operate

1 ctrl+z: 暂停（中断）仿真，使用fg来恢复；

2 ctrl+c: 停止仿真；

3 ctrl+a: 鼠标移动到敲写命令的开头；

4 ctrl+e: 鼠标移动到敲写命令的结尾.

### 11） makefile/shell/perl/python

IC designer需要掌握的4大脚本工作。

如果执行脚本的方式是make run，那么很可能你用到的是一个Makefile脚本；如果执行方式是source run，那么这应该是一个Shell语言写成的脚本；如果是其它情况，那么就得看具体这个脚本首行是怎么写的了。Makefile和Shell语言比Perl/Python要更容易上手，写起来也更加简单，比较适合满足一些非常简单的批量任务需求。Perl的强项则在于它强大的文本处理能力和无所不能的CPAN库，随时可以满足你的各种任性需求。Python的优点则是较好的可维护性。

### 12）FIFO min depth

异步FIFO最小深度常用计算公式

写时钟频率w\_clk,

读时钟频率 r\_clk,

写时钟周期里，每B个时钟周期会有A个数据写入FIFO

读时钟周期里，每Y个时钟周期会有X个数据读出FIFO

计算公式如下：

fifo\_depth = burst\_length - burst\_length \* X/Y \* r\_clk/w\_clk

推导过程：

首先，我们可以认为写操作是Burst突发的。

其次，写操作的效率并不是100%的，而是A/B的，因此我们可以认为实际的F\_wr = (A/B)\*w\_clk，同理，实际中F\_rd = (X/Y)\*r\_clk。

另外，这里面并没有约束Burst突发的场景，在正常情况下，应该是这样的

空闲---Burst突发---空闲---Burst突发---空闲---Burst突发。

所以我们在计算中，需要考虑最极端的情况：

空闲---Burst突发---Burst突发---空闲---Burst突发---空闲。

即传输过程中，可能会出现"背靠背"的情况，那么我们设计的FIFO深度必须能够保正，在"背靠背"的时间段内，如果接收方没法接受所有数据，那么剩余的数据可以被存储在FIFO内部且不会溢出。

假设"背靠背"时发送的数据 = BL，那么"背靠背"的时间 = BL / w\_clk ，注意，这段时间内 F\_wr = w\_clk，而不是之前提到的 (A/B)\*w\_clk。

在这段时间内，接收方可以接受的数据 = (BL / w\_clk) \* (X/Y)\*r\_clk ，

剩下的数据量 = BL - ( BL / w\_clk ) \* (X/Y)\*r\_clk，那么FIFO的深度至少就要为：

depth = BL - ( BL / w\_clk ) \* (X/Y)\*r\_clk

将上述公式变换下，得到 depth = BL - BL \* (X/Y) \* (r\_clk/w\_clk) 。

如果100个写时钟周期可以写入80个数据，10个读时钟可以读出8个数据。令wclk＝rclk ，考虑背靠背（20个clk不发数据＋80clk发数据＋80clk发数据＋20个clk不发数据的200个clk）代入公式可计算FIFO的深度:

fifo\_depth = 160-160X（80%）=160-128=32

如果令wclk＝200mhz，改为100个wclk里写入40个，rclk＝100mhz，10个rclk里读出8个。那么fifo深度为48, 计算如下:

fifo\_depth =80-80X（80%）X(100/200)=80-32=48

### 13) Makefile

gcc编译器有许多选项，一般来说我们只要知道其中的几个就够了。

-o选项我们已经知道了，表示我们要求输出的可执行文件名。 -c选项表示我们只要求编译器输出目标代码，而不必要输出可执行文件。 -g选项表示我们要求编译器在编译的时候提供我们以后对程序进行调试的信息。

比如：

gcc -o hello hello.c

gcc 编译器就会为我们生成一个hello的可执行文件。执行./hello就可以看到程序的输出结果了。命令行中 gcc表示我们是用gcc来编译我们的源程序，-o 选项表示我们要求编译器给我们输出的可执行文件名为hello 而hello.c是我们的源程序文件。

Makefile有三个非常有用的变量。分别是$@，$^，$<代表的意义分别是：

$@--目标文件，$^--所有的依赖文件，$<--第一个依赖文件。

## 3 Verilog HDL

### 1） Previous pendings and skills

1 function 不能调用task，但task可以彼此调用，还可以调用function。另外，task里面不可以有always这样的语句，亦即task即使实现某种逻辑功能，也只能实现组合逻辑功能（如果写成module就不局限于此）。

task和function综合出来的电路都是组合电路，如果你想在task和function来写时序功能，那你需要的是module而不是这两个；task和function的区别是很明显的，基本上都是写法上，定义上，调用上的区别。根据这些区别，很容易辨识他们的应用场景

2 coding style，注意阻塞与非阻塞赋值不乱用。ex：

~~（updata axi\_salve model后，run checklist，发现waveform的波形没有拉起来，看起来是GMINT design的request没有拉起来，但考虑到之前axi\_slave model没有问题，所以我认为问题应该出在updata后axi\_slave.v model上，通过diff两个新旧model，采用了比较笨的办法，将新model中更改的部分一步步改回来，最后定位到这里：）~~

always @ (count or wready\_e or swt\_wptr or swt\_rptr)

if(axis\_pipe==1)begin

if(count == 0)begin

s\_AWREADY <= `DELAY (~s\_AWREADY) && (!swt\_full);

s\_ARREADY <= `DELAY (~s\_ARREADY) && (!srd\_full);

在组合逻辑中采用了非阻塞赋值，最后即使不报出ERROR，波形也不会符合预期。

3 assign clk\_o = sel\_clkb ? clkb : clka

sel\_clkb必须是一个DFF输出信号.

4、格雷码的产生：

assign nxt\_wptr\_wclk = (af\_wr\_en && !af\_full) ? (wptr\_wclk + 1'b1) : wptr\_wclk ;

assign nxt\_wptr\_gray = (nxt\_wptr\_wclk >> 1) ^ nxt\_wptr\_wclk ;

### 2） Basic Rule

#### 1、亚稳态

所有的亚稳态，归根结底就是setup/hold时间不满足导致。在同一个时钟域下的信号，综合以及布线工具可以在data路径或者clock路径上插入buffer使得每一个DFF的setup/hold时间都满足。

消除亚稳态，就是采用多级DFF来采样来自另一个时钟域的信号，级数越多，同步过来的信号越稳定。对于频率很高的设计，建议至少用三级DFF，而两级DFF同步则是所有异步信号处理的最基本要求。

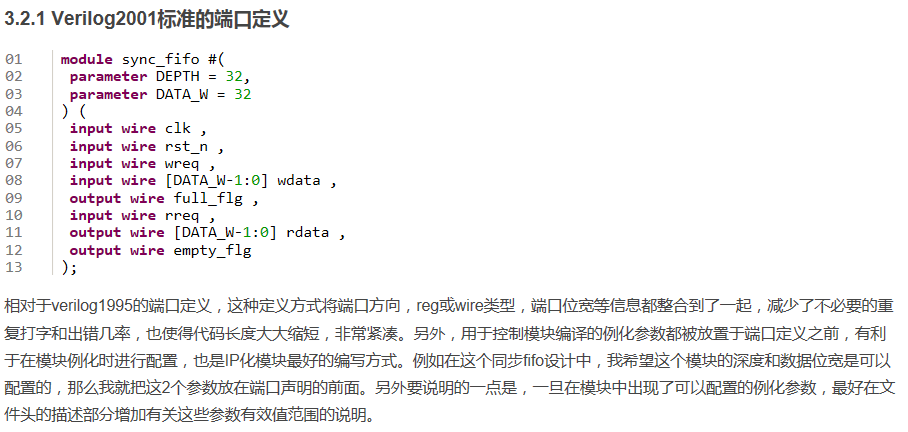
#### 2、脉冲信号跨时钟

clka下的脉冲信号，同步到clkb时钟域下，它对于clka与clkb的时钟频率关系没有任何限制，快到慢，慢到快都没问题。其主要原理就是先把脉冲信号在clka下展宽，变成电平信号，再向clkb传递，当确认clkb已经“看见”信号同步过去之后，再清掉clka下的电平信号。

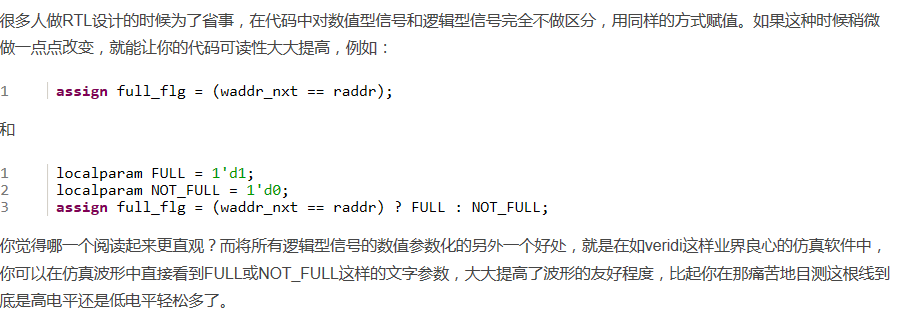
#### 3、FIFO

FIFO用于为匹配读写速度而设置的数据缓冲buffer，当读写时钟异步时，就是异步FIFO。多bit的数据信号，并不是直接从写时钟域同步到读时钟域的，而是读写时钟域分别派遣了一个信使，去通知对方时钟域，当前本方所处的读写情况，来判断还能不能写以及可不可以读，这两个信使就是读写指针。

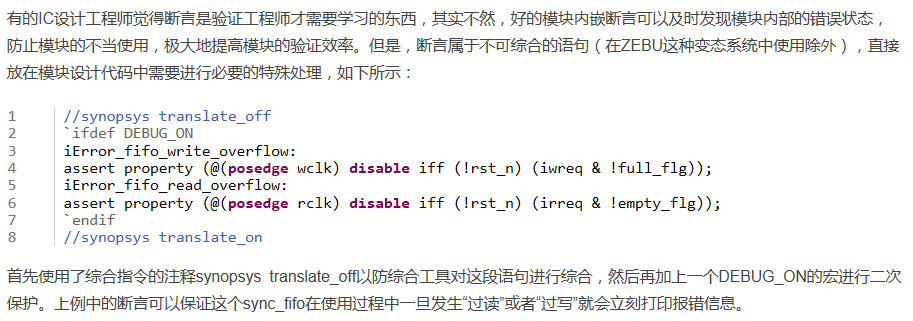
#### 4、 verilog2001标准的端口定义：



#### 5、 逻辑型信号采用参数赋值：



#### 6、 内嵌断言



#### 7、竞争冒险：

多个信号经过不同路径到达某一点有时间差，不能完全sync，称为竞争；

由于竞争使得design产生了暂时的错误输出，称之为冒险。

竞争冒险由组合逻辑产生。

有竞争不一定有冒险，但出现了冒险就一定存在竞争.

#### 8、 always块中的clock

clk最好不要同时上下沿采样：

在一个always中，一般不允许同时posedge clk or negedage clk，如果分开， 两个always，按说是没有绝对意义上的错误，但一般不建议同时使用negedge clk，这样对时序要求很高，也容易出现时序问题；

#### 9、产生随机数函数：

系统函数$random(seed)时，可以写成三种样式：1）$random，2）$random()，3）$random(seed)。下面分别说明：

1）$random

这是最简单的一种写法，略去了seed这个传入参数，$random会使用一个默认的seed（这个默认值为0？）。也正因此，每次进行仿真时，$random产生的随机数序列都是相同的。

2）$random()

这种写法和写法1）的作用是相同的，同样是没有给$random传入seed。

3）$random(seed)

这种写法与上面两种不同，给$random传入了参数seed，因此$random根据seed来产生随机数。seed不同，产生的随机数的序列也不同。而且，每执行一次$random(seed)产生一个随机数，seed也自动更新一次。

比如：rand\_rank={$random(seed)}% 4; 随机产生0, 1, 2, 3.

seed的产生和修改：

产生方法两种：

1 initial块中直接赋值（建议）；

integer seed;

initial  begin

seed = 0;

end

2 initial块中 seed = $get\_initial\_random\_seed()；

修改的两种方法：

1 修改代码，在第一种产生seed的基础上；

2 采用deposit方式；

这种方式比较灵活，不必修改代码，也不必重新编译，直接修改输入到仿真软件的命令即可。

下面是最简单的一个命令：

deposit top.seed 2

run 125ns

exit

上面命令的意思是，将信号（或变量）seed的初值设成1，然后开始仿真，仿真时间为125ns，然后结束仿真。

$random和{$random}的区别：

reg[23:0] rand;

rand=$random%60; //产生一个在 -59—59范围的随机数

rand={$random} %60; //通过位拼接操作{}产生0—59范围的随机数

产生特定范围随机数的技巧：

# 在6到30之间产生随机数.

rnumber=$((RANDOM%25+6))

# 还是在6-30之间产生随机数,

#+但是这个数还必须能被3整除.

rnumber=$(( RANDOM%27/3\*3+6))

其他随机函数：

$dist\_uniform(seed,start,end);

比如下面这些代码，用来产生0，1的随机数：

randomwr=$dist\_uniform(seed,0,1);

case(randomwr)

'h0:mxuw\_test;

'h1:mxur\_test;

endcase

### 3） SPYGLASS

1 RTL编写时最好width一致，比如：

assign d\_SEQIDI[4:0] = d\_RDQ\_RPTR ? d\_RDQ1SID[4:0] : d\_RDQ0SID[4:0];

否则会报warning；

2

### 4） define，parameter和localparam

`define：常用于定义常量可以跨模块、跨文件; 范围 -> 整个工程;

一旦`define指令被编译，其在整个编辑过程中都有效，例如，通过另一个文件中的`define指令，定义的常量可以被其他文件调用，直到遇到`undef;

parameter：常用于模块间参数传递; 范围 -> 本module内有效的定义;

localparam ： 常用于状态机的参数定义;范围 -> 本module内有效的定义，不可用于参数传递; localparam cannot be used within the module port parameter list.一般情况下，状态机的参数会用到localparam，或者某些逻辑型信号：

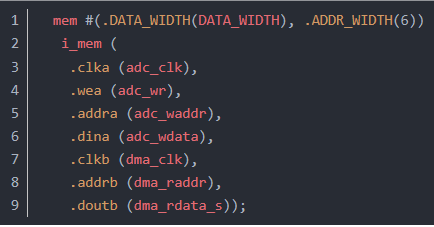
localparam Full = 1’b1;

localparm Empty = 1’b0;

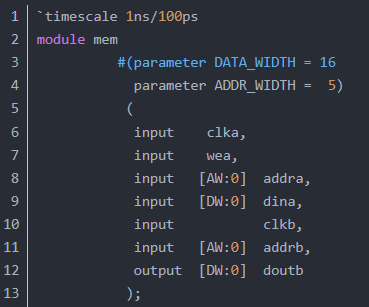
assign Full\_flag = (wrptr = rdptr)? Full:Empth;

ex:

顶层文件或header文件的参数例化：



底层文件的参数定义（在moduel端口定义之前）--optional



### 5） Primitive（UDP）

UDP以保留字primitive（原语）开始，以endprimitive结束，并紧接着原语的Ports/terminals。这与module的定义类似。UDP应该定义在module和endmoudle外面。

For example:

primitive mux21hd\_bdddrdqs\_s13 (out, sel, i0, i1);

output out;

input sel, i0, i1;

table

0 0 ? : 0 ;

0 1 ? : 1 ;

1 ? 0 : 0 ;

1 ? 1 : 1 ;

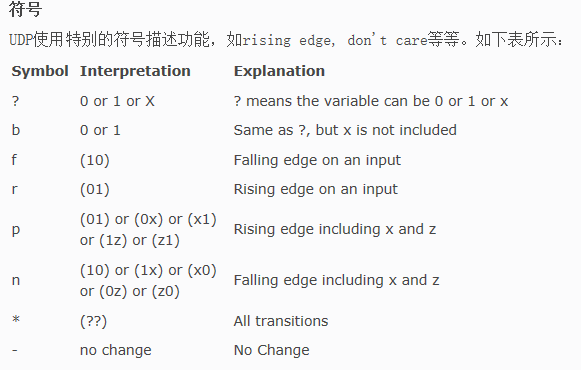
? 0 0 : 0 ;

? 1 1 : 1 ;

endtable

endprimitive

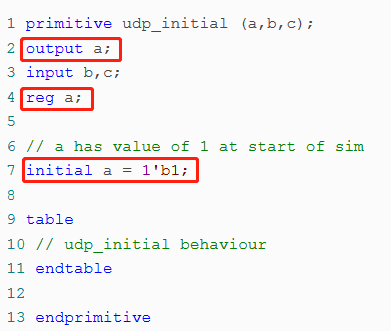
tabe中的符合定义：



端口的要求：

* 一个UDP只可以包含一个输出和最多10个输入。
* 输出端口应该是第一个端口，然后才是一个或多个输入端口。
* 所有的UDP都是标量，也就是，向量端口不允许。
* UDP不能是双向端口。
* 时序UDP的输出端需要额外声明为reg类型。
* 组合UDP的输出端声明为reg类型是非法的。

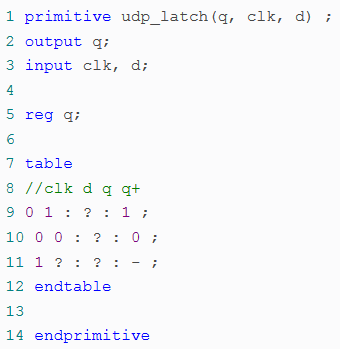
功能体：原语的功能（包括组合和时序）在一个table表里描述，以保留字endtable结束。对于时序UDP，我们可以使用initial 给输出赋一个初始值，比如：



时序UDP - 电平敏感：

电平敏感时序行为的描述与组合行为相同（除了输出需要声明为reg类型，且在table中有一个额外的区域，用于代表当前UDP的状态）。

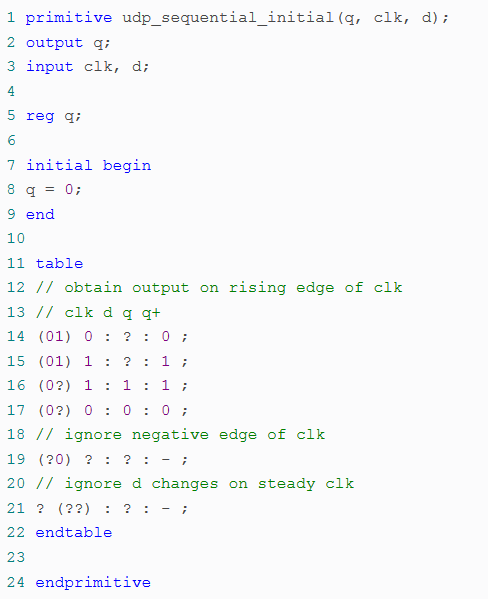
输出声明为reg表示UDP有一个内部状态，UDP的输出总是与内部状态相同。时序UDP与组合UDP比起来，在输入和输出区域之间多了一个额外的区域，代表当前状态区域，且当前状态与当前输出值相同，这个区域也用冒号和输入输出分隔开。



该示例是一个低电平敏感的锁存器，q即是当前输出也是当前状态，q+即为下一个输出，表现和组合电路类似。

时序UDP - 边沿敏感：

和电平敏感的输入值和当前状态决定输出值不同，边沿敏感是由其输入的变化触发的输出改变。如下图所示：



不影响输出的所有转换必须显性的指定。否则，他们将导致输出值变为x。如果UDP对任何输入的边沿敏感，对所有输入的所有边沿都必须指定需要的输出状态。

### 5） Specify

specify block用来描述从源点（source：input/inout port）到终点（destination：output/inout port）的路径延时（path delay），由specify开始，到endspecify结束，并且只能在模块内部声明，具有精确性（accuracy）和模块性（modularity）的特点。specify block可以用来执行以下三个任务：

一、描述横穿整个模块的各种路径及其延时。（module path delay）

二、脉冲过滤限制。（pulse filtering limit）

三、时序检查。（timing check）

specify block有一个专用的关键字specparam用来进行参数声明，用法和parameter一样，不同点是两者的作用域不同： specparam只能在specify block内部声明及使用，而parameter只能在specify block外部声明及使用。

module path delay：

1、simple path：

1)、并行连接（patallel connection）：source => destination  
2)、全连接（full connection）：      source \*> destination  
例：    (a, b => q, qn) = 1; 等价于：  
            (a => q) = 1; (b => qn) = 1;  
而(a, b \*> q, qn) = 1; 等价于：  
            (a => q) = 1; (b => q) = 1; (a => qn) = 1; (b => qn) = 1;

2、edge sensitive path：是那些源点（source）使用边沿触发的路径，并使用边缘标示符指明触发条件（posedge/negedge），如果没有指明的话，那么就是任何变化都会触发终点（destination）的变化。

1）：(posedge clk => (out +: in)) = (1,2);

在clk的上升沿，从clk到out的模块路径，其上升延时是1，下降延时是2，从in到out的数据路径是同向传输，即out = in。

2）：(negedge clk => (out -: in)) = (1,2);

在clk的下降沿，从clk到out的模块路径，其上升延时是1，下降延时是2，从in到out的数据路径是反向传输，即out = ~in。

3）：(clk => (out : in)) = (1,2);

clk的任何变化，从clk到out的模块路径，其上升延时是1，下降延时是2，从in到out的数据路径的传输是不可预知的，同向或者反向或者不变。

Note：模块路径的极性（module path polarity）：未知极性（unknown polarity，无），正极性（positive polarity，+），负极性（negative polarity，-）。

3、state dependent path：是那些源点（source）以来指定条件状态的路径，使用if语句（不带else）。在条件=1 or X or Z的情况下，认为条件成立。如果有一条路经，存在多个条件同时成立的情况，那么使用延时最小值的那条限制。

1）： specify

        if(a)     (b => out) = (1,2);

        if(~a)    (b => out) = (2,3);

        if(b)     (a => out) = (1,2);

        if(~b)    (a => out) = (2,3);

      endspecify

2）： specify

        if(rst)   (posedge clk => (q +: data)) = (1,2);

        if(~rst)  (posedge clk => (q +: data)) = (2,3);

      endspecify

需要注意的是，所有输入状态都应该说明，否则没有说明的路径使用分布延时（distributed delay），如果也没有声明分布延时（distributed delay）的话，那么使用零延时（zero delay）。如果路径延时和分布延时同时声明的话，则选择最大的延时作为路径延时。另外，也可以使用ifnone语句，在其它所有条件都不满足的情况下，说明一个缺省的状态依赖路径延时。

3）： specify

        (posedge clk => (q +: data)) = (1,2);

        ifnone (clk => q) = (2,3);

      endspecify

pulse filtering limit：

由于每条传播路径都具有一定的电容性和电阻性，电荷无法在一瞬间积累或消散，所以信号变化的物理特性是具有惯性的。为了更准确地描述这种能力，使用惯性延时（inertial delay），它可以抑制持续信号比传播延时短的输入信号的变化。

timing check:

描述设计要求的时序性能，所有的时序检查有一个reference event和一个data event, 两者通过一个布尔表达式相连接，还包括一个可选的notifier register选项，用来打印错误信息或者传播x状态。

## 4 VIP DDR5

### 1) DDR5 VIP ENV

  DDR5 VIP license 只能到9/30, 请基于下面的方法，拿一个例子熟悉下吧

 1 DDR5 VIP 拿的方法：

  % $DESIGNWARE\_HOME/bin/dw\_vip\_setup -i home ---用这个命令可以去加载ddr example, 如：

ddr\_svt/tb\_ddr5\_dimm\_svt\_uvm\_basic\_sys

ddr\_svt/tb\_ddr5\_dimm\_svt\_ovm\_basic\_sys等

% $DESIGNWARE\_HOME/bin/dw\_vip\_setup -path <design\_dir> -e ddr\_svt/tb\_ddr5\_dimm\_svt\_uvm\_basic\_sys -svlog

<design\_dir> 你要加载的路径：

2在你拿到的环境里面路径下会有

examples/sverilog/ddr\_svt/tb\_ddr5\_dimm\_svt\_uvm\_basic\_sys

这个是VIP 环境的内部一个例子

3 run的方法

ts.backdoor\_memory\_memh\_test.sv

ts.backdoor\_memory\_mif\_test.sv

ts.catalog\_rdimm\_test.sv

ts.catalog\_udimm\_rw\_test.sv

ts.power\_up\_initilization\_test.sv

ts.randomized\_dimm\_configuration\_rw\_test.sv   ---这些是环境里面带的test文件，一共有6条

run的方法如下：

      For example, to run test ts.backdoor\_memory\_mif\_test.sv, do following:

      gmake USE\_SIMULATOR=vcsvlog backdoor\_memory\_mif\_test WAVES=fsdb

and run ts.catalog\_udimm\_rw\_test.sv, do following:

gmake USE\_SIMULATOR=vcsvlog catalog\_udimm\_rw\_test WAVES=fsdb

这样跑完之后会有一个test\_top.fsdb 这个是波形文件

env.dimm\_memory.channel\_\*.data\_lane\_\*.rank\_\*.monitor.cmd\_trace 这个是在仿真过程中向rank\* data\_lane\* channel\* 发了什么cycle,可以结合波形看看

### 2) 三种DIMM

目前内存条的种类有三种：UDIMM, RDIMM, LRDIMM。

1.UDIMM：也称为Unbuffered DIMM。当数据从CPU传到每个内存颗粒时，UDIMM要求保证CPU到每个内存颗粒之间的数据传输距离相等，这样并行传输才会有效。这需要极高的制造工艺，极难做到高密度、高频率。因此UDIMM容量和频率都较低。不过，UDIMM由于在CPU和内存之间没有任何缓存，因此同频率下时延较小。目前常见的是单条容量2GB/4GB，最高主频也只能到达1.33GHz。

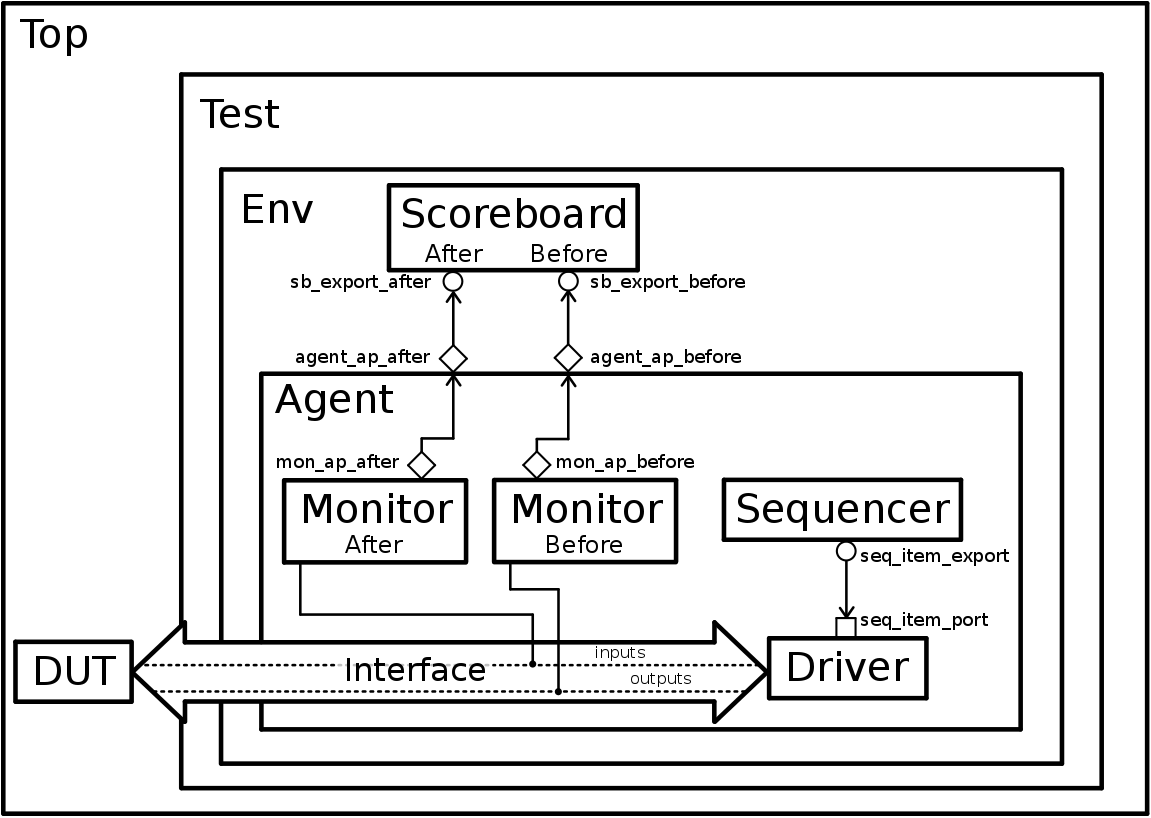
2.RDIMM：也称为Registered DIMM。为了保证并行传输的有效性，RDIMM在内存条上加了一个寄存器进行转发。它位于CPU和内存颗粒之间，这样就减少了并行传输的距离。同时由于寄存器效率很高，因此RDIMM的密度和频率就容易提高。RDIMM目前是较为主流的内存条，单条容量在2～32GB之间，频率也有1.33GHz和1.6GHz两种选择。绝大多数2路通用配置的服务器出厂时通常都会配置这种类型的内存.

3.LRDIMM：也称为Load Reduced DIMM。RDIMM虽然提高了传输有效性，但由于寄存器大小有限，当单条内存中内存颗粒以最高的密度4 Rank进行部署时，并行传输的有效性和频率就会大大下降。例如在使用32G RDIMM时，在服务器的每个内存通道上最多只能部署2条内存条，而且只能运行在800MHz。 LRDIMM内存通过将当前RDIMM内存上的Register芯片改为一种iMB（isolation Memory Buffer）内存隔离缓冲芯片来降低内存总线的负载，并相应地进一步提升内存支持容量。相比于通常的RDIMM，Dual-Rank LRDIMM内存的功耗只有其50%，Quad-Rank LRDIMM也能低到其75%。目前，典型的Nehalem-EP处理器可以支持3个内存通道，每个内存通道最多支持3个RDIMM，而改用LRDIMM内存之后，同样的系统可以每通道支持到9个DIMM，内存容量提升到原来的三倍。

SPD: 串行存在检测，不需要BIOS侦测内存，通过BIOS读通SPD来取得内存的相关信息。

## 5 UVM

UVM包括transaction、interface、driver、sequence、sequencer、monitor、reference model、agent、test、env、top等部分，其相互关系极其复杂。不如说，UVM牺牲简洁性换来“通用”性.



https://www.cnblogs.com/bettty/p/5285785.html

## 6 English Note

### 1) terminology(术语)

1 waive 丢掉，舍弃，忽略

2 promote: 促进，提升。ex：当GMINT这路的DMAXI 很久没有得到arbitration，他会自己time out尽快得到arbitration ，但是并不会promote自己的high priority。

3 symbolic 象征的，常用-s表示，用于linux命令，如ln –s，建立软链接。

4 debounce电路，就是常说的去抖滤波，主要用在芯片的PAD输入信号，或者模拟电路输出给数字电路的信号上。

5 outstanding: 未解决的，可以理解我pending的东西。

ex:

GFX can only send out one outstanding MXU\_GMI\_FLH\_REQ to GMINT before GMI\_MXU\_FLH\_ACK back.;

Tracing Q is designed as 64 level(the level is related to the max outstanding read request for DRAMC

6 correspond to: 与……相一致

ex:

Target ID with the back data, it corresponds to engine source MXU\_GMI\_SID[4:0].

7 nested:嵌套的

A block comment shall start with /\* and end with \*/. Block comments

shall not be nested.

8 Unary operators：一元运算符，如+，-，\*，/。

Unary operators shall appear to the left of their operand.

9 decimal\_number | octal\_number | binary\_number | hex\_number

十进制，八进制，二进制，十六进制

10 DDR5 spec terminology words:

interval：间隔； be bounded to:有界限的，约束于； swap:交换；

interchange:互换；coincident:一致； strap value: 捆绑值，类似于case语句选择；

mandatory:强制的，托管的； ramp:斜坡，坡道； duration:持续时间；

stable: 稳定的，牢固的； packet: 数据包； payload:有效负荷;

impedance:阻抗；subsequence:后续的，后继的； concurrent: 并发的，一致的；

lockout: 停工，封锁； engaged:忙碌的，使用中的； consecutive:连贯的；

seamless: 无缝的； precedence: 优先；retrieve:检索，恢复；

identical: 同一的，完全相同的；conjunction:结合，连接，同时发生；drift:漂流，漂移；

halt:停止，立定，犹豫； retention: 保留，滞留； statically:静态的，静止的；

complies(comply):遵从，依从； deviate: 脱离，越轨；spectrum: 光谱，频谱，范围；

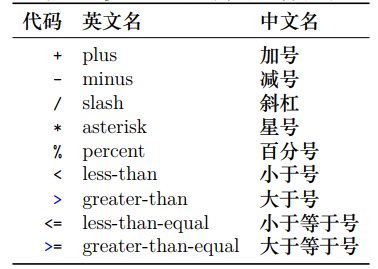
sole: 单独的，仅有的； extent: 程度，范围； exceed: 超过，胜过；

crosstalk: 串扰； restore: 恢复，修复； accomplished: 完成的，有技巧的，熟练；

coarse: 粗俗的，粗糙的； instantaneous: 即时的，瞬时的； accumulated:累积的；

likewise: 同样地，也；alias: 别名，化名； mandatory: 强制的，托管的；

seasoned:经验丰富的，老练的； mandate：授权； undesirable：不合需要的



### 2) Sentence(术语)

Sorry, the subscriber you dialed is busy now, please redial later.

The legends say that “Bug” comes from an actual moth（飞蛾） that flew into one of the first computers causing it to malfunction（发生故障）.