Copyright by SmartLogic , 2017



UCP1.0 SDK使用手册

目录

[第一部分工具链 4](#_Toc505075690)

[1 汇编器 4](#_Toc505075691)

[1.1 汇编器llvm-mc 4](#_Toc505075692)

[1.2 标量处理器SPU的汇编用法 5](#_Toc505075693)

[1.3 标量处理器SPU的反汇编用法 5](#_Toc505075694)

[1.4 向量处理器MPU的汇编用法 5](#_Toc505075695)

[2 汇编的高级用法 6](#_Toc505075696)

[2.1 汇编语言中的宏 6](#_Toc505075697)

[2.1.1 宏的申明和定义 6](#_Toc505075698)

[2.1.2 宏的调用和展开 6](#_Toc505075699)

[2.1.3 宏的参数使用 7](#_Toc505075700)

[2.1.4 宏的嵌套和递归 8](#_Toc505075701)

[2.2 微码工具中的横向宏 8](#_Toc505075702)

[2.2.1 横向宏的定义 9](#_Toc505075703)

[2.2.2 横向宏的调用 9](#_Toc505075704)

[2.2.3 横向宏的一些使用约定 12](#_Toc505075705)

[2.2.4 用横向宏实现流水线 12](#_Toc505075706)

[3 链接器 15](#_Toc505075707)

[3.1 命令格式 15](#_Toc505075708)

[3.2 命令选项 15](#_Toc505075709)

[3.3 用例 15](#_Toc505075710)

[4 编译器clang 16](#_Toc505075711)

[4.1 命令格式 16](#_Toc505075712)

[4.2 命令选项 16](#_Toc505075713)

[4.3 用例 17](#_Toc505075714)

[第二部分模拟器 17](#_Toc505075715)

[5 UCP1.0模拟器简介 17](#_Toc505075716)

[5.1 UCP模拟器功能特性 17](#_Toc505075717)

[6 UCP1.0模拟器的在线调试功能 18](#_Toc505075718)

[6.1 UCP模拟器的调试功能 18](#_Toc505075719)

[6.2 UCP模拟器调试注意事项 18](#_Toc505075720)

[7 调试界面使用说明与示例 19](#_Toc505075721)

[7.1 调试界面组成 19](#_Toc505075722)

[7.2 调试流程 20](#_Toc505075723)

[8 版本修改记录 24](#_Toc505075724)

# 第一部分工具链

# 汇编器

## 汇编器llvm-mc

llvm-mc是UCP的汇编器，用于将汇编文本文件生成二进制文件。

llvm-mc的使用格式如下：

llvm-mc [options] <input file>

列出了llvm-mc常用的命令选项。

表1llvm-mc常用的命令选项

|  |  |
| --- | --- |
| 选项名称 | 用途 |
| -arch=[mspu|mmpu|ucps32|ucpm48] | 指定指令集体系结构，其中mspu和mmpu是MaPU1.0的spu和mpu的体系结构，ucps32和ucpm48是UCP1.0的spu和mpu的体系结构 |
| -o | 指定输出文件名 |
| -I | 指定包含文件的路径 |
| -g | 生成调试信息 |
| -filetype=[asm|obj] | 指定输出文件的格式，缺省为asm |
| -help | 获取更多的使用帮助 |

示例：下述命令实现汇编功能：

|  |
| --- |
| llvm-mc -arch=ucpm48 -filetype=objhello.mpu.s -o hello.o |

## 标量处理器SPU的汇编用法

UCP1.0的标量处理器SPU的体系结构参数是“ucps32”，汇编用法如下(这里spu汇编程序源文件推荐使用.s后缀，后缀之前的文件名可以采取符合通用命名规范的名称即可)：

llvm-mc -arch=ucps32 -filetype=objhello.spu.s -o hello.o

## 标量处理器SPU的反汇编用法

llvm-objdump是spu的反汇编器，用于将spu二进制文件（elf文件）生成汇编文本文件。

表2列出了llvm-objdump常用的命令选项。

表2llvm-objdump常用的命令选项

|  |  |
| --- | --- |
| 选项名称 | 用途 |
| -arch=[mspu|mmpu|ucps32] | 指定指令集体系结构 |
| -d [-disassemble] | 反汇编 |
| -r | 输出重定向信息 |
| -g | 输出行信息 |
| -t | 输出符号表 |
| -no-show-raw-insn | （可选）反汇编时不打印指令的字节 |
| -help | 获取更多的使用帮助 |

下述命令实现反汇编功能：

|  |
| --- |
| llvm-objdump -arch-name=ucps32 hello.o–d -no-show-raw-insn |

## 向量处理器MPU的汇编用法

UCP1.0的标量处理器MPU的体系结构参数是“ucpm48”，汇编用法如下(这里mpu汇编程序源文件推荐使用.s后缀，后缀之前的文件名可以采取符合通用命名规范的名称即可)：

llvm-mc -arch=ucpm48 -filetype=objhello.mpu.s -o hello.o

# 汇编的高级用法

## 汇编语言中的宏

宏是汇编语言中提供的一种类似于C语言#define的功能。它可以有效地提高代码的可复用性，进而提高开发效率。本节以GNU AS工具中的宏为例，介绍宏的定义、调用和其他特性。

### 宏的申明和定义

宏的申明和定义由directive关键字标识，其结构如下：

**.macro** identifier parameterlist

macro body

**.endmacro**

其中第一行是宏的申明，identifer是申明宏的名称，parameterlist是宏的参数申明部分。

macro body是宏的定义部分，可以包含除.endmacro以外的所有汇编指令和directive。

.endmacro标识宏的定义结束，也可以用.endm简写。

### 宏的调用和展开

宏的最大作用是提高代码的复用性和编程效率，调用宏可以用宏的名称代替宏定义中内容。每一次调用相当于在调用处做一次宏名称和宏定义的代码替换。

宏的调用格式是在代码行的起始处使用宏的名称，以及（如果存在）传递参数的列表。如：

**.macro** example

BIU1.T0(A++)->W1.M[3](Mode0);

IFALU.(T0,T0) IND T1(T1)->IFMAC.T0(Mode0);

**.endmacro**

NOP;

example

NOP;

example

MPUSTOP;

等价于：

NOP;

BIU1.T0(A++)->W1.M[3](Mode0);

IFALU.(T0,T0) IND T1(T1)->IFMAC.T0(Mode0);

NOP;

BIU1.T0(A++)->W1.M[3](Mode0);

IFALU.(T0,T0) IND T1(T1)->IFMAC.T0(Mode0);

MPUSTOP;

### 宏的参数使用

在宏中引入参数的目的是对宏定义中的一些代码做文本替换。宏的参数有三种定义方式：

1用,分隔：argu0, argu1, …

2用空格分隔：argu0 argu1 …

3带默认值的定义：argu0=0, argu1 …

参数在宏定义中用前缀“\”标识，如：\argu0。

**.macro**load from, to, t\_reg=0, flags

.if \flags

BIU\from .DM (K++)(A++) ->\to. T\t\_reg;

.else

BIU\from .DM ->\to. T\t\_reg;

.endif

**.endmacro**

load 0, FALU, , 1

load 1, FALU, 1, 0

等价于：

BIU0 .DM(K++)(A++) ->FALU. T0;

BIU1 .DM ->FALU. T1;

### 宏的嵌套和递归

宏支持嵌套定义，即在一个宏定义中可以调用其他的宏。此外，宏可以在定义中对自身调用，称为递归嵌套。

嵌套可以进一步提高代码的可重用性以及代码的层次化结构。

在实例化时，宏会从最外层开始按先后顺序逐层展开。注：在宏定义中实例化的宏不会被立即展开，要等到最外层宏实例化的时候才会被展开。

## 微码工具中的横向宏

虽然宏具有很好的可重用性和可配置性，但是宏只能实现文本的替代，而且这种替代只能是纵向按行替换。换而言之，同一行内不可能同时出现两个宏的内容，从而无法将两个宏的内容横向拼接起来。

微码工具的横向宏提供了一种横向拼接宏的机制。横向宏在实例化时是按列展开，每个横向宏被展开成一列，多个横向宏形成多个列的文本拼接起来。在微码工具中，横向宏除了做简单的文本拼接外，还支持状态机的微码程序拼接。

### 横向宏的定义

横向宏的定义在语法格式上与宏类似，如下所示：

**.hmacro** identifier

instructions

**.endhmacro**

横向宏不支持参数，对于需要宏参数功能的横向宏，可以通过将横向宏定义在宏定义内部的形式实现，例如：

**.macro**ld no, len, mask

**.hmacro**ld\no

FALU.T\no (I\no)=BIU\no .DM(A++)(K++);REPEAT @(\len);;

**.if** \mask

FALU.T\no (I\no)=BIU\no .DM(M);;

**.endif**

**.endhmacro**

**.endmacro**

注：实例化的时候要先实例化一次ld宏，产生横向宏ld\no的定义，再实例化横向宏ld\no。

横向宏的定义通常由微码指令组成，不支持大多数的directive，目前支持的directive包括：.if、.else、.endif等。

### 横向宏的调用

由于横向宏是按列拼接，并且横向宏关注宏定义内部的控制流，所以横向宏的调用和展开情况比宏要复杂。在不同的上下文中调用横向宏会得到不同的结果。如前所述，横向宏不但能够实现文本的横向拼接，还能识别不同横向宏的控制流，进行状态机拼接，因此，横向宏展开分为两种方式：按文本拼接和按状态机拼接。文本拼接时，忽略横向宏中MSEQ槽里的指令含义；状态机拼接时，按照第二章所述的微码程序拼接规则进行拼接。给出了微码工具展开横向宏的规则和示例。

表3展开横向宏的规则和示例

|  |  |  |  |
| --- | --- | --- | --- |
| 上下文 | 前后文 | 展开说明 | 示例 |
| 在横向宏定义中 | 单独出现 | 相当于与一个NOP同行，按照状态机拼接 | .hmacroexmp  IALU.T0(I0)=BIU0.DM;REPEAT @(3);;  .endhmacro  .hmacro main  exmp;;  REPEAT @(2);;  endhmacro  main;;  等价于：  IALU.T0(I0)=BIU0.DM;;  IALU.T0(I0)=BIU0.DM;REPEAT @(2);; |
| 与REPEAT指令同行 | 相当于当前横向宏与一个REPEAT块拼接，按照状态机拼接 | .hmacroexmp  IALU.T0(I0)=BIU0.DM;REPEAT @(3);;  .endhmacro  .hmacro main  exmp;FALU.T0(I1)=BIU1.DM;REPEAT @(2);;  endhmacro  main;;  等价于：  IALU.T0(I0)=BIU0.DM;FALU.T0(I1)=BIU1.DM;REPEAT @(2);;  IALU.T0(I0)=BIU0.DM;; |
| 与横向宏同行 | 两个横向宏按照状态机拼接 | .hmacro exmp0  IALU.T0(I0)=BIU0.DM;REPEAT @(3);;  .endhmacro  .hmacro exmp1  REPEAT @(2);;  .endhmacro  .hmacro main  exmp0;exmp1;;  .endhmacro  main;;  等价于  IALU.T0(I0)=BIU0.DM;REPEAT @(2);;  IALU.T0(I0)=BIU0.DM;; |
| 不在横向宏定义中 | 单独出现 | 相当于与一个NOP同行，按照文本拼接 | .hmacroexmp  IALU.T0(I0)=BIU0.DM;REPEAT @(3);;  IALU.STOP;;  .endhmacro  exmp;;  REPEAT @(2);;  等价于：  IALU.T0(I0)=BIU0.DM;REPEAT @(3);;  IALU.STOP;REPEAT @(2);; |
| 与微码指令同行 | 按照文本拼接，可能存在控制槽冲突 | .hmacroexmp  IALU.T0(I0)=BIU0.DM;;  IALU.STOP;;  .endhmacro  exmp;REPEAT @(2);;  等价于：  IALU.T0(I0)=BIU0.DM;REPEAT @(2);;  IALU.STOP;; |
| 与横向宏同行 | 两个横向宏按照状态机拼接 | .hmacro exmp0  IALU.T0(I0)=BIU0.DM;REPEAT @(3);;  .endhmacro  .hmacro exmp1  REPEAT @(2);;  .endhmacro  exmp0;exmp1;;  等价于  IALU.T0(I0)=BIU0.DM;REPEAT @(2);;  IALU.T0(I0)=BIU0.DM;; |

### 横向宏的一些使用约定

标号的使用：在横向宏中只允许使用数字标号，数字标号是一种局部临时标号，在引用标号时需要在标号后面加上“b”或者“f”表示向上或者向下查找最近的匹配数字标号。在最终生成的代码或符号表中，以“HM”开头的标号为工具生成的标号，所以在微码程序中“HM+数字”形式的标号是保留标号，不要手动使用。

KI的修正：由于状态机的拼接会做指令提取，而提取指令会提取循环体，从而使原来重复KI次的循环体需要少循环若干次，而在工具自动拼接时，这个次数是工具编译时计算得到的，所以会以符号赋值的形式表现出来。符号的名称格式为：KInCALIBRm，其中n表示KI的编号，m表示对KIn的第m次出现进行的修正。

横向宏的结束：由于横向宏包含多行内容，而调用时只占用一行，所以在源文件结束时，横向宏中的指令可能还没有发射完，因此，在使用横向宏的源文件中用.mpustop告诉工具不再向下读取源文件，并将未发射的横向宏按顺序和拼接规则发射完。

### 用横向宏实现流水线

横向宏可以有效地解决流水线上分立状态机的拼接困难。在开发组合状态机的微码程序时，先把各个独立状态机的微码程序定义成横向宏，再通过按延迟调用横向宏实现状态机组成的流水线。

以2.4节中的例子，说明横向宏实现一个等价的流水线。

BIU0横向宏：

**.hmacro**biu0\_hm

LPTO 1f @(KI0);;

SHU0.T0=BIU0.DM (A++)(K++);;

REPEAT @(15);;

1:

**.endhmacro**

SHU0横向宏：

**.hmacro**shu0\_hm

LPTO 1f @(KI0);;

FMAC.T1(I1)=SHU0.T0 IND T1(TB =+ 4);;

FMAC.T1(I1)=SHU0.T0 IND TB(TB += 4);;REPEAT @(15);;

1:

**.endhmacro**

流水线横向宏：

**.hmacro**pipeline

biu0\_hm;;

REPEAT @(7);;

shu0\_hm;;

**.endhmacro**

由于SHU0的状态机相对于BIU0有8个周期的延迟，所以在BIU0的状态机启动后插入7个NOP。由于biu0\_hm;;相当于biu0\_hm;NOP;;，所以程序可以简化为：

**.hmacro**pipeline

biu0\_hm;REPEAT @(8);;

shu0\_hm;;

**.endhmacro**

由于是在横向宏定义的上下文中，所以REPEAT @(8)不会和biu0\_hm做文本拼接，而是做状态机拼接。此外，简化后的程序对于工具而言会产生跟简练的拼接微码程序。

工具的拼接结果：

NOP ;;

SHU0.T0 = BIU0.DM (A++) (K++) ;;

REPEAT @( 6 ) ;;

KI0CALIBR0 = -1

LPTO HM1 @( KI0 ) ;;

FMAC.T1 (I1) = SHU0.T0 INDX T1 (TB =+ 4) ;;

FMAC.T1 (I1) = SHU0.T0 INDX TB (TB += 4) ; REPEAT @ ( 7 ) ;;

FMAC.T1 (I1) = SHU0.T0 INDX TB (TB += 4) ; SHU0.T0 = BIU0.DM (A++) (K++) ;;

FMAC.T1 (I1) = SHU0.T0 INDX TB (TB += 4) ; REPEAT @ ( 6 ) ;;

FMAC.T1 (I1) = SHU0.T0 INDX TB (TB += 4) ;;

HM1:

FMAC.T1 (I1) = SHU0.T0 INDX T1 (TB =+ 4) ;;

FMAC.T1 (I1) = SHU0.T0 INDX TB (TB += 4) ; REPEAT @ ( 7 ) ;;

FMAC.T1 (I1) = SHU0.T0 INDX TB (TB += 4) ; REPEAT @ ( 8 ) ;;

# 链接器

ld.gold是spu和mpu的链接器，用于链接spu、mpu的二进制文件。同时，spu的链接 clang工具来实现，见第一节用法。

## 命令格式

ld.gold [options] file...

## 命令选项

列出了ld.gold常用的命令选项。

表4ld.gold常用命令选项

|  |  |
| --- | --- |
| 选项名称 | 用途 |
| -T | 指定链接脚本 |
| -l | 指定搜索库 |
| -L | 指定搜索库所在的路径 |
| --just-symbols | 只链接符号 |
| -d | 为链接过程定义符号 |
| -o | 指定输出文件的名称 |
| -help | 获取更多的使用帮助 |

## 用例

下述命令实现链接功能：

|  |
| --- |
| ld.goldhello.o -o hello |

下述命令指定链接脚本：

|  |
| --- |
| ld.gold-T ld.sc crt.ohello.o -o hello |

下述命令将一个spu程序与mpu程序链接起来，这种使用方式最常见于在spu程序中调用mpu的代码。

|  |
| --- |
| ld.gold-T ld.sc spu.o--just-symbolsmpu.o -o hello |

# 编译器clang

clang是spu的编译器，用于将C程序编译生成UCP体系结构的汇编文件，可重定位文件，以及可执行文件等。

## 命令格式

clang[options] <input object files>

## 命令选项

下表列出了llvm-objdump常用的命令选项。

|  |  |
| --- | --- |
| 选项名称 | 用途 |
| --target=ucps32 | 指定指令集体系结构 |
| -E | 预处理 |
| -S | 预处理+编译 |
| -c | 预处理+编译+汇编 |
| 空选项 | 预处理+编译+汇编+链接 |
| -o | 指定输出文件 |
| -v | 打印详细命令信息 |
| -g | 生成源代码级调试信息 |
| -help | 获取更多的使用帮助 |

## 用例

下述命令实现预处理+编译功能：

|  |
| --- |
| clang –S--target=ucps32 -o hello.shello.c |

下述命令实现预处理+编译+汇编功能：

|  |
| --- |
| clang –c--target=ucps32 -o hello.ohello.c |

下述命令实现预处理+编译+汇编+链接功能：

|  |
| --- |
| clang --target=ucps32 -o hello.outhello.c |

# 第二部分模拟器

# UCP1.0模拟器简介

## UCP模拟器功能特性

UCP模拟器是一款用C++语言开发的UCP内核指令集模拟器。其实现了对UCP内核体系结构中标量处理单元SPU和向量处理单元MPU的所有指令的功能模拟，并对所有指令的执行周期进行了精确建模。其主要功能是提供快速仿真的、与硬件行为一致的虚拟仿真平台，供软件开发人员调试程序使用。

UCP模拟器主要特性如下：

* 与硬件一致的SPU和MPU同步执行
* 严格按照所有指令的行为描述进行了功能建模
* 周期精确的SPU和MPU流水线建模。除存储器访问冲突情况之外，SPU和MPU所有指令的执行周期与硬件执行周期完全一致
* 包含了指令存储器IM、MIM和数据存储器DM0~DM2的行为级模型
* 快速的仿真。相比于RTL仿真，可得到1000X倍以上的加速性能

# UCP1.0模拟器的在线调试功能

## UCP模拟器的调试功能

UCP模拟器需要与调试界面配合使用，以实现在线调试。模拟器需要的输入文件包括：在SPU中执行程序的二进制代码文件IM.data，在MPU中执行程序的二进制代码文件MIM.data，DM0~DM2中的初始化数据二进制代码文件DM0.dat、DM1.dat和DM2.dat。关于上述输入文件如何生成，参照第一部分工具链使用说明文档。

算法程序运行完，模拟器将会在算法代码所在路径下输出五个.dat文件。其中，Simulator\_DM0\_Result.dat、Simulator\_DM1\_Result.dat、Simulator\_DM2\_Result.dat存放程序运行后DM0~DM2中的数据，Simulator\_IM\_Result.dat存放SPU程序存储器数据，Run\_Cycle.dat存放算法程序的总执行周期数。

## UCP模拟器调试注意事项

UCP模拟器环境运行如下：

* 操作系统：64位Linux3.10.0
* 64位QT 5.9.1
* GNUlibc 2.17

QT安装完成后，如果出现错误：

error while loading shared libraries: libQt5Widgets.so.5: cannot open shared object file: No such file or directory

解决方法如下：

1. 在文件/etc/ld.so.conf末尾添加动态库的路径，例如: /opt/Qt\_5.9.1/5.9.1/gcc\_64/lib
2. 执行sudoldconfig命令

DM0.dat、DM1.dat和DM2.dat文件必须先指明数据存储起始地址，再接有效数据。地址表示方式为：以“@”开头，后跟5位十六进制数，例如：“@7ffc0”。

# 调试界面使用说明与示例

## 调试界面组成

调试界面包含以下几个组成部分：

如图1，右边区域为主界面部分，包括SPU Code, MPU Code, Memory view, Register四个部分。

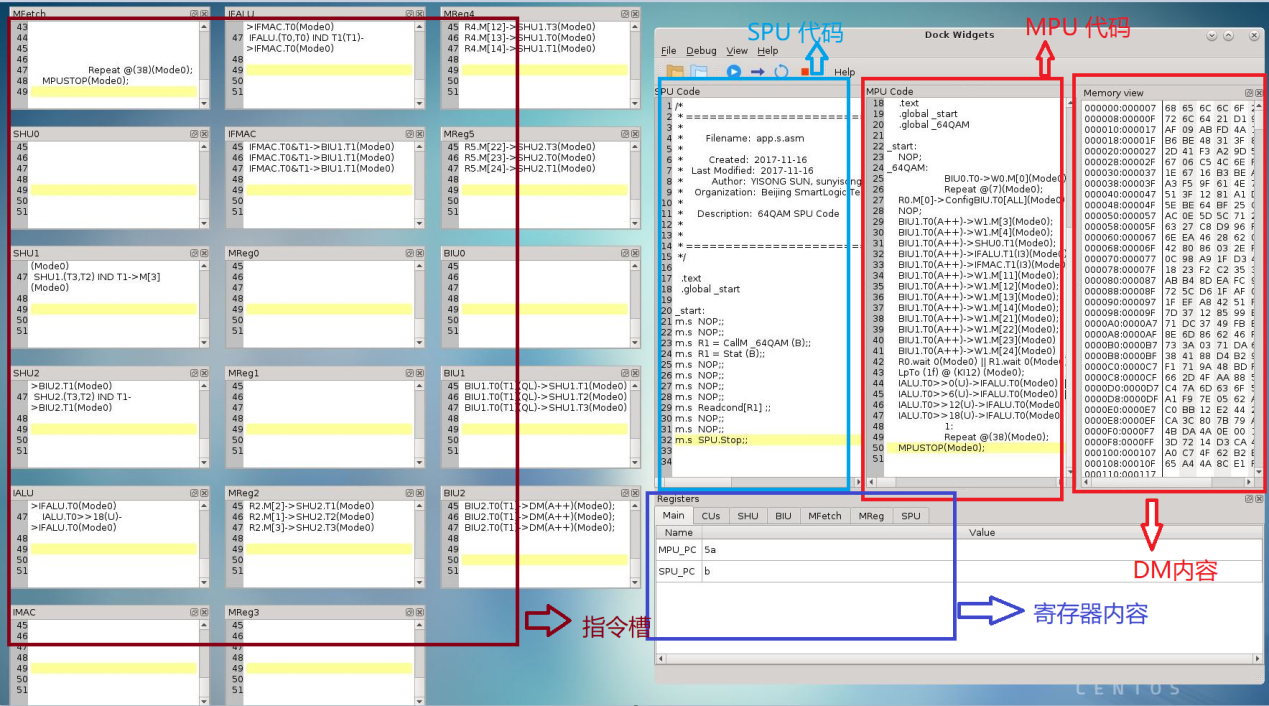


图1. 模拟器界面

其中：

1. SPU Code：负责显示SPU代码，并且可以通过点击左侧灰色数字栏设置/取消断点。
2. MPU Code：负责显示MPU代码，也可以通过点击数字栏设置/取消断点。
3. Registers: 负责显示不同功能部件中寄存器的数值。其有七个Main, CUs,SHU,BIU,MFetch, MReg, SPU七组部件。
4. Memory View:显示DM中数据，可以通过右击输入地址查询对应地址内容。此窗口在初始时不显示，程序开始运行后显示。

Registers与Memory View中的数值随着程序运行而实时更新。

其中左边区域为17个浮动窗口，每一个窗口显示对应指令槽中指令。窗口中黄色高亮部分表示即将执行的指令。显示浮动窗口的快捷键为 Ctrl + S。

## 调试流程

以64QAM算法程序的调试过程为例，对调试过程的演示说明如下：

1. 通过Linux系统命令行窗口启动模拟器： ./UCP\_Simulator
2. 启动界面程序DebGuiR1，如图2所示。

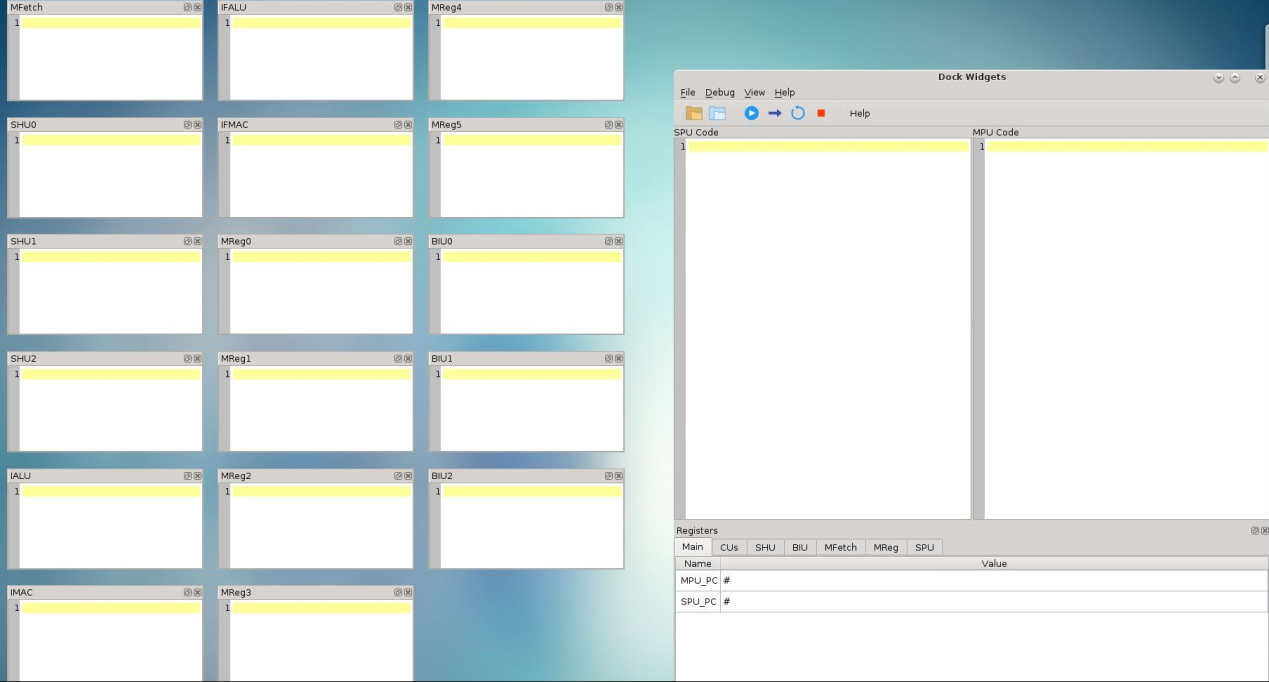


图2. 程序启动后界面

1. 读入SPU程序，如图3所示

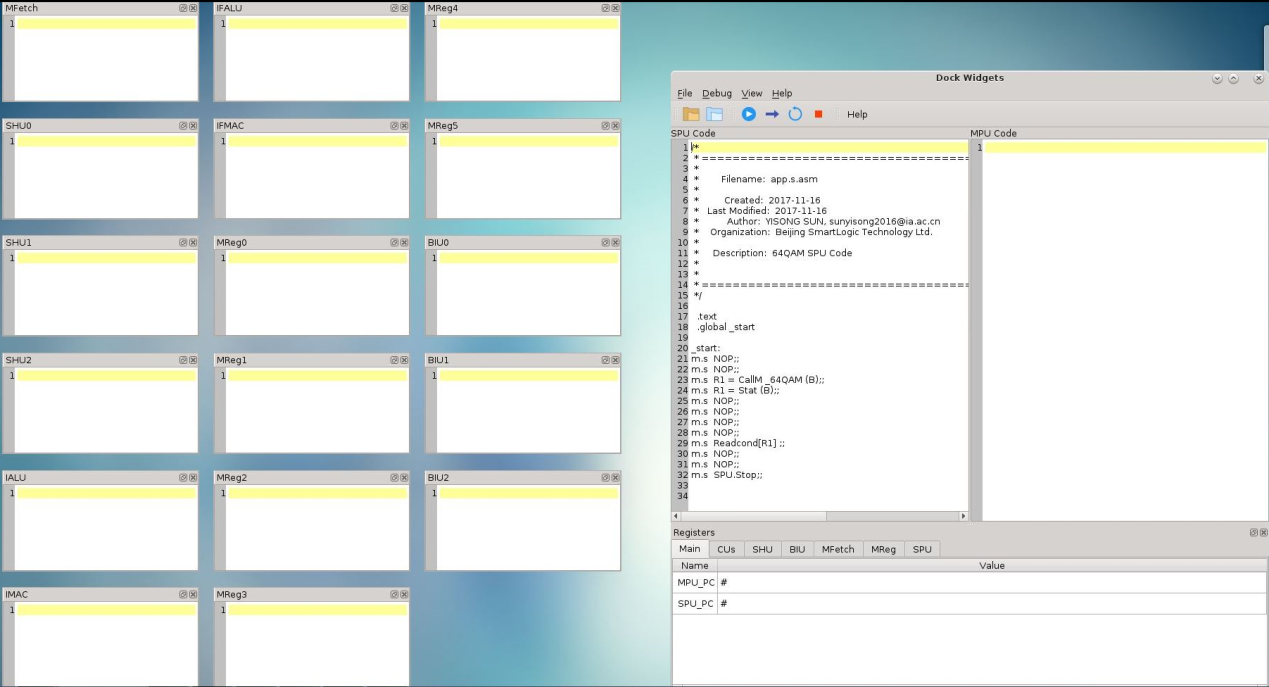


图3. 读入SPU代码

1. 读入MPU代码，如图4所示。注意此时左侧浮动窗口已经读入对应指令（因为前面部分cycle其为空指令，故显示空白，但行号已经发生改变）



图4 读入MPU代码

1. 设置断点。如图5，在SPU和MPU中分别加入一个和两个断点（图中红色），可以通过单击取消



图5. 设置断点

1. 点击蓝色Run标志，此时执行到第一个断点处。

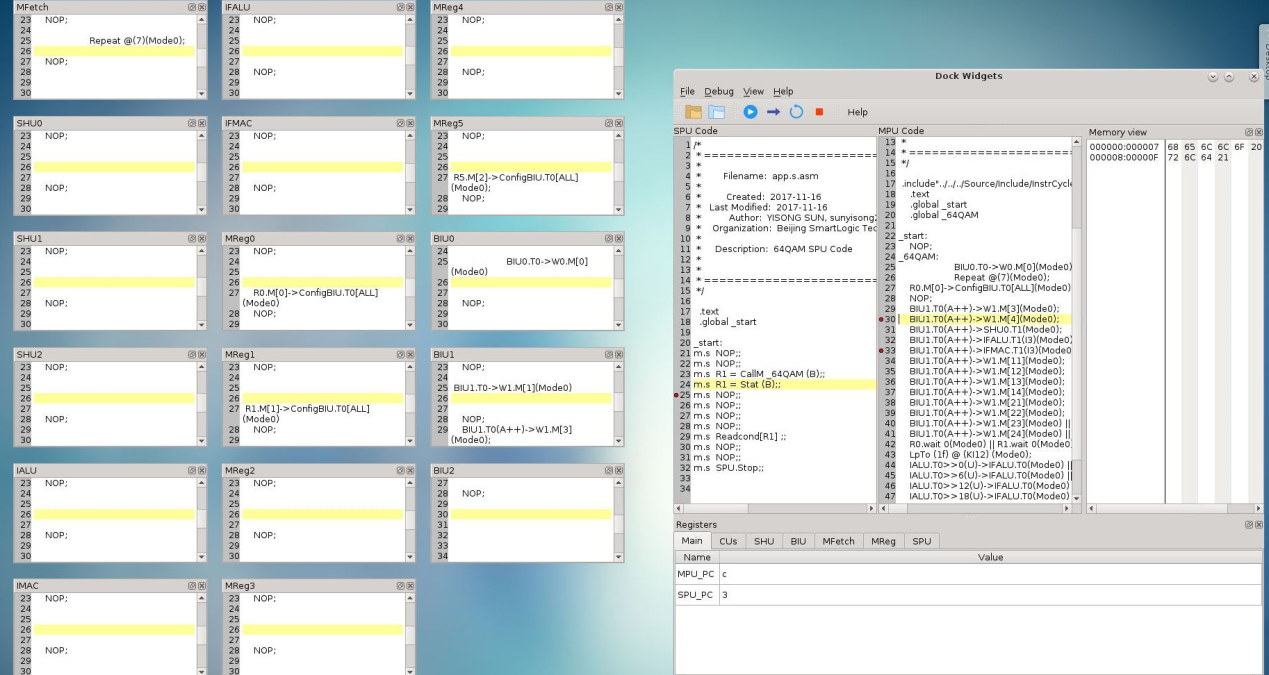


图6. Run执行到第一个断点处

1. 点击Step进行单步执行，如图6，图7所示，为点击两次Step对应界面。

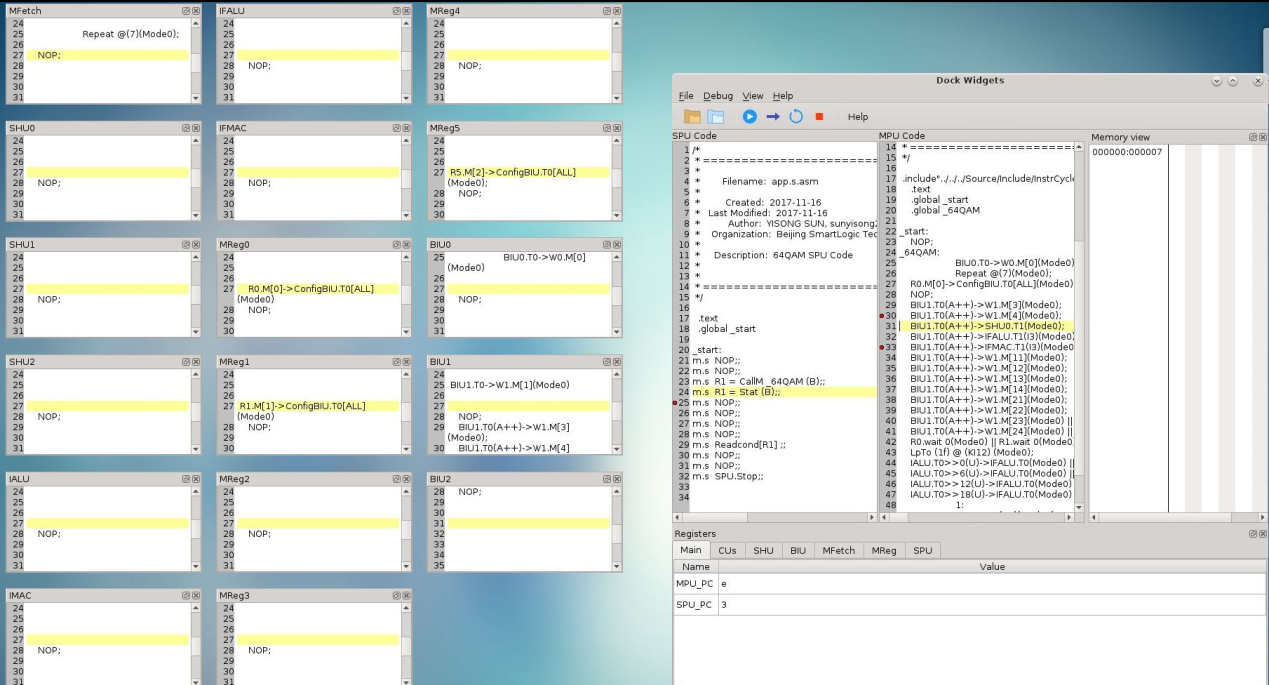


图7. 单步执行Step

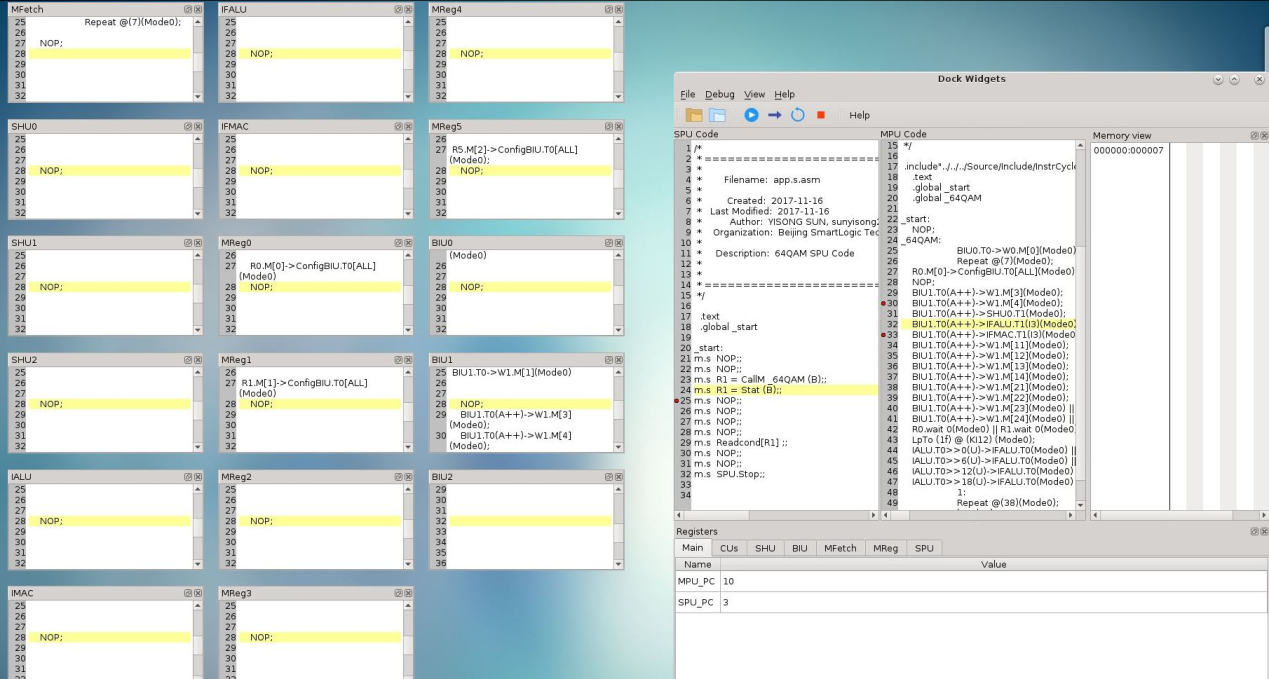


图8. 单步执行

1. 执行到最后，并右击Memory View区域输入地址查询DM中内容。

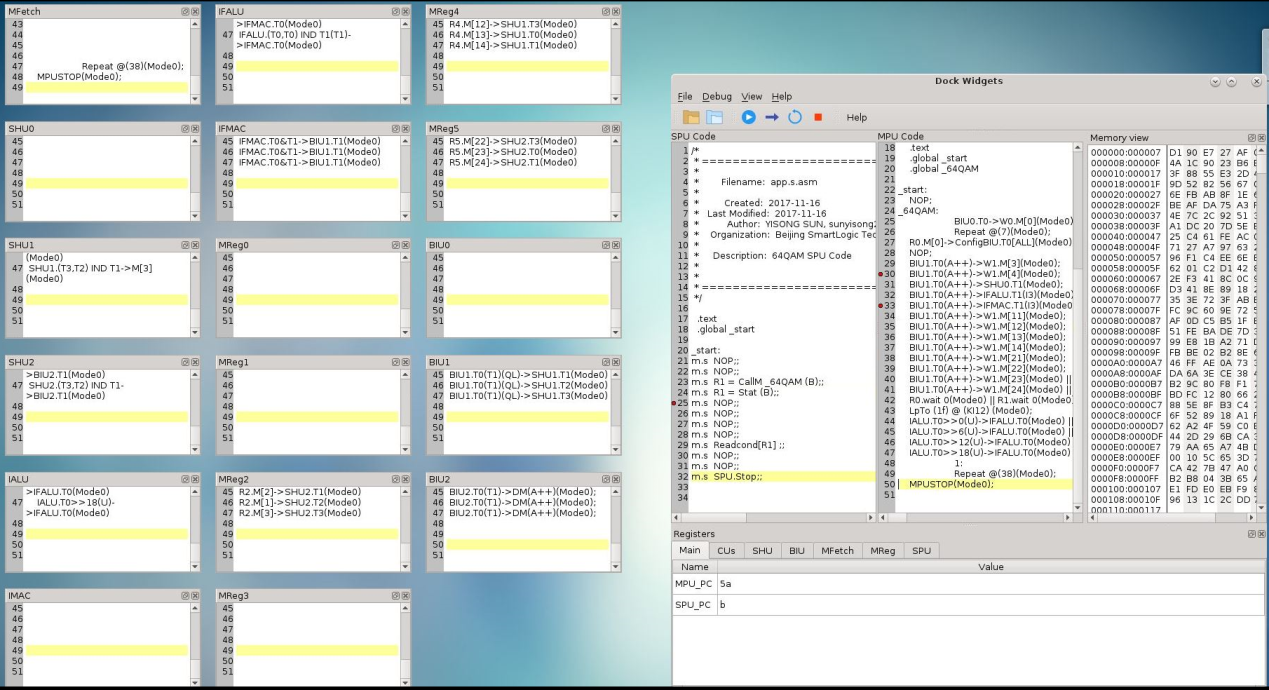


图9 执行到最后

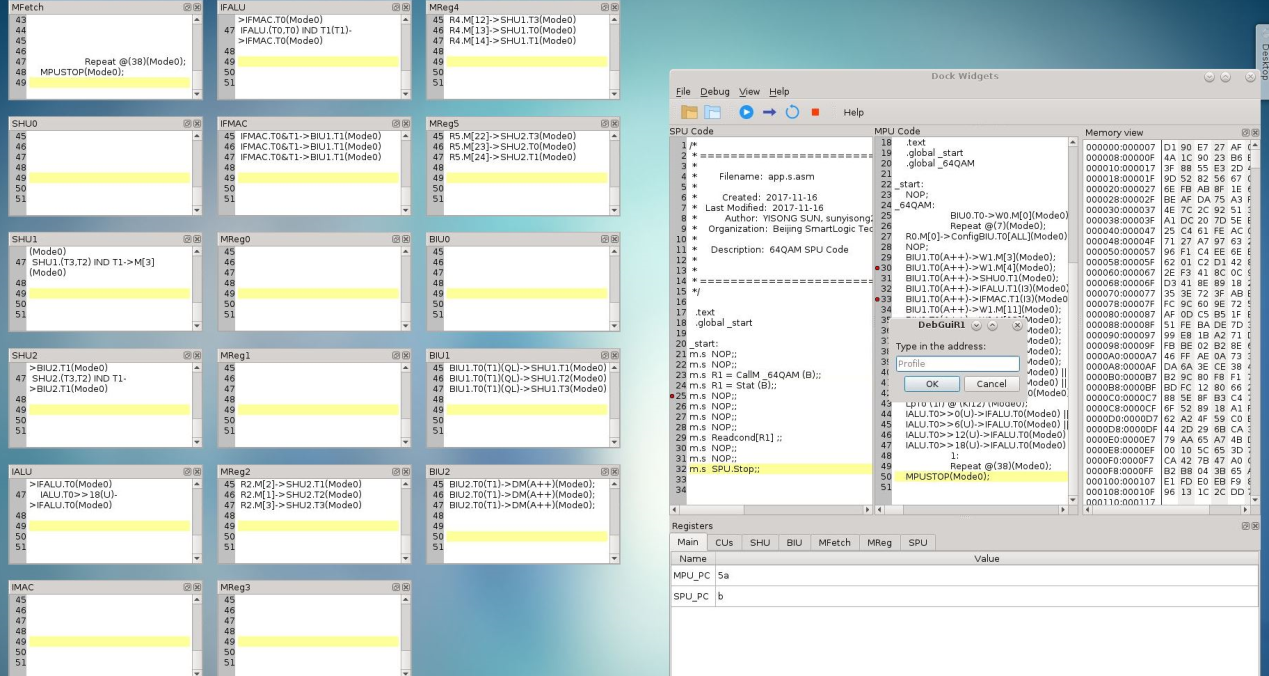


图10 查询DM

# 版本修改记录

版本号说明：版本号分为三个部分，分别表示发行版本、功能性修改版本和描述性修改版本，中间用小数点隔开，即(发行版本.功能性修改版本.描述性修改版本)，如(1.2.11)。其中发行版本指本次公开发布的版本号，功能性修改版本指对系统功能定义进行了修改，需描述性修改版本指系统功能没有变，但功能描述发生了变化(如错误字、措辞更改等)。

|  |  |  |
| --- | --- | --- |
| 版本号 | 审阅/修改日期 | 审阅意见/修改说明 |
| 0.1. | 2017-10-19 | 创建版本 |
| 0.2 | 2017-11-19 | 添加UCP1.0工具链的使用方法说明 |
| 0.3 | 2017-12-19 | 添加UCP1.0模拟器的使用方法说明 |
| 1.0 | 2018-01-05 | 修改标量处理器SPU的反汇编命令；  添加UCP1.0模拟器的调试注意事项 |
| 1.1 | 2018-01-12 | 添加模拟器调试输出文档说明；添加QT安装配置说明；添加调试界面高亮显示说明 |
| 1.2 | 2018-01-19 | 修改模拟器调试界面组成中，关于Memory View数值显示、更新的描述 |
| 1.3 | 2018-01-30 | 添加UCP1.0编译器clang使用方法说明 |