实验11：  
探究存储器层级

北卡罗莱纳州立大学  
电子与计算机工程系

Alexander G. Dean博士  
2016年1月7日，版本1.1



**

*内容*

[概述 4](#_Toc462326578)

[设置 4](#_Toc462326579)

[打开和配置项目 4](#_Toc462326580)

[下载代码 4](#_Toc462326581)

[程序评估 5](#_Toc462326582)

[源代码 5](#_Toc462326583)

[外部时序分析 6](#_Toc462326584)

[目标代码 6](#_Toc462326585)

[减少闪存等待状态 8](#_Toc462326586)

[使用预取模块 9](#_Toc462326587)

[CPU指令访问 9](#_Toc462326588)

[CPU指令和数据访问 9](#_Toc462326589)

[所有存储器访问 10](#_Toc462326590)

[使用高速缓存 10](#_Toc462326591)

[使能和清除高速缓存 10](#_Toc462326592)

[更改数组大小 11](#_Toc462326593)

[更改数组访问模式 11](#_Toc462326594)

# 概述

在本实验练习中，您将通过使能存储器层级中的提速功能来加快程序的运行。

与许多其他高性能CPU一样，PIC32MZ单片机中的MIPS CPU执行指令的速度要远快于其访问存储器的速度。对于200 MHz系统时钟速率，PIC32MZ每5 ns可完成一条指令，而访问存储器则会花费较长的时间（例如，对于闪存程序存储器，需要15 ns）。这种存储器访问延时由两或三部分组成：将请求从CPU经由互连系统发送至存储设备、访问存储单元，以及将数据经由互连系统返回到CPU（针对读取操作）。

为防止这种“存储困境”显著降低程序运行速度，已向存储器系统添加多个性能加速功能。在本实验中，我们将研究预取模块和高速缓存。

本实验支持使用PIC32MZ2048**EF** MCU和PIC32MZ2048**EC** MCU的Wi-FIRE开发板。本实验使用基本I/O屏蔽来显示时序结果。

答案以红色斜体（MySolution）样式呈现。在分发给学生前，应删除这些部分。此外，显示的时间适用于EF处理器。两个处理器的时间和图形包含在Lab11\_Timing\_Analysis\_Solutions.xlsx电子表格中。

# 设置

## 打开和配置项目

1. 使用6引脚插座或电缆将PICkit 3调试器插入chipKIT板的JP1（ICSP）中。确保PICkit 3调试器上的白色三角指向JP1的引脚1（方形焊盘）。
2. 在chipKIT板上安装基本I/O屏蔽。
3. 使用USB电缆将PICkit 3调试器和chipKIT Wi-Fire板连接到PC。
4. 在PC上启动MPLAB X IDE程序。
5. 在MPLAB X IDE的**“File”（文件）**菜单中，选择**“Open Project…”（打开项目…）**，然后选择名为**Lab11\_Cache**的项目。
6. 单击“扫帚和锤子”图标（位于MPLAB X IDE窗口的顶部）以清除并编译主项目。   
   
7. MPLAB X IDE右下角窗格中的输出选项卡将显示这些操作的进度和结果。请注意消息“BUILD SUCCESSFUL”（编译成功），它指示的是总体状态。

## 下载代码

1. 现在，您可以将程序下载到chipKIT中。单击绿色向下箭头图标（位于MPLAB X IDE窗口的顶部）完成此操作。   
    
2. PICkit3上的状态指示灯将在编程期间呈红色和绿色闪烁状态，然后在完成时保持绿色常亮状态。“Output”（输出）窗口将打开名为**“PICkit 3 debugger”（PICkit 3调试器）**的选项卡以显示编程状态。

# 程序评估

为评估CPU和存储器系统，我们将使用一个可在位于SRAM中的二维整数数组中累加所有元素的程序。数组的维数由NUM\_ROWS和NUM\_COLS定义，为方便起见，将NUM\_COLS设为与NUM\_ROWS相等的值。

## 源代码

数组求和函数的源代码如下所示。代码的执行时间使用定时器TMR2测量，此定时器每4个CPU时钟周期（每20 ns）递增一次。

unsigned Timed\_Array\_Test\_CR(int nr, int nc, int a[][nc], int \* ret\_sum) {

int r, c;

int sum = 0;

unsigned t;

BIOS\_LD1\_PORT\_BIT = 1;

TMR2 = 0;

r = 0;

do {

c = 0;

do {

sum += a[r][c];

c++;

} while (c < nc );

r++ ;

} while (r < nr);

t = TMR2;

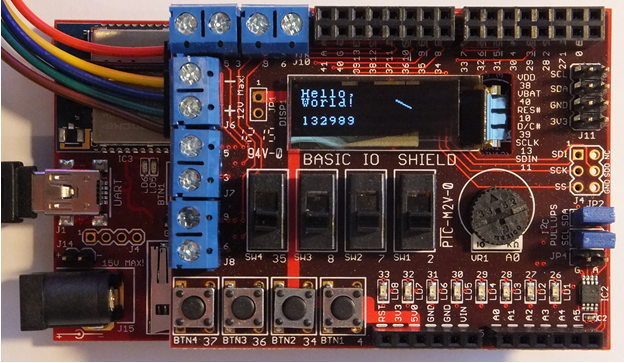
BIOS\_LD1\_PORT\_BIT = 0;

\* ret\_sum = sum;

return t;

}

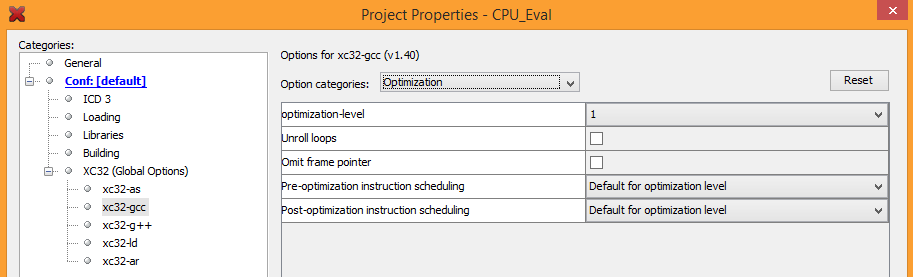
## 外部时序分析



验证代码的时序行为以确保定时器已正确配置是一种好习惯。可使用示波器来观察执行   
Timed\_Array\_Test\_CR所花费的时间。代码会在执行定时操作时点亮基本I/O屏蔽的LED LD1。可借助示波器或逻辑分析器来测量此时间，方法是将其探头置于LD1（标签“26”）一侧。

## 目标代码

确保编译器gcc已设置为优化级别1（-O1），如项目属性中所示。这将有助于编译器快速生成代码。



数组求和函数中嵌套循环的目标代码如下所示，为方便理解，其中添加了注释。源文件的目标代码位于项目目录的build\<配置>\production\下。

Timed\_Array\_Test\_CR:

… ; entry code removed for clarity

move $8,$0 ; sum = 0

move $10,$0 ; r = 0

.L5: ; Outer Loop (over r)

move $3,$6 ; pointer to array a

move $2,$0 ; c = 0

.L3: ; Inner Loop (over c)

lw $9,0($3) ; load a[r][c] from memory

addu $8,$8,$9 ; add to sum

addiu $2,$2,1 ; c = c + 1

slt $9,$2,$5 ; set $9 if more columns (c < nc)

bne $9,$0,.L3 ; repeat loop .L3 if more columns

addiu $3,$3,4 ; advance pointer to next element in array’s column

addiu $10,$10,1 ; r = r + 1

slt $2,$10,$4 ; set $2 if more rows (r < nr)

bne $2,$0,.L5 ; repeat loop .L4 if more rows

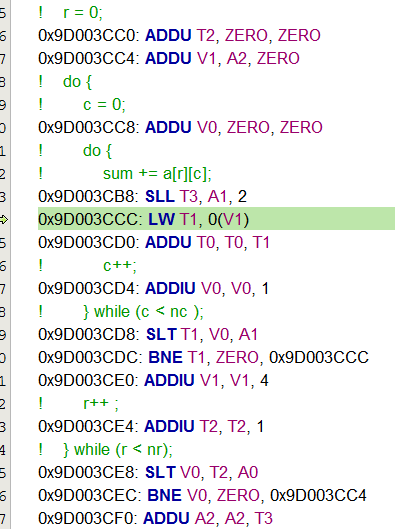
addu $6,$6,$11 ; advance pointer to next row’s element in array

… ; exit code removed for clarity

请注意，内循环（以绿色显示）有六条指令，因此我们预计代码至少会占用六个CPU周期。请记住，由于指令执行流水线的原因，分支指令（bne）会被延时并在下一条指令（addiu）*后*执行。

代码中存在数据冲突：装入字指令（lw）的结果由指令（addu）使用，这会导致一个周期的延时。因此，我们预计代码的执行至少需七个周期。

请注意，以上所示代码与CPU执行的代码不***完全***相同。汇编器可能会略微优化或更改代码。我们可以使用调试器的反汇编程序查找确切的代码，如下所示。我们发现，伪指令move实际上是通过add指令实现的*。*地址0x9D003CB8处的sll指令由编译器的优化活动移动。



# 减少闪存等待状态

CPU从闪存中取指令以执行程序。如果CPU的时钟速率（fsys）为66 MHz或更小值，则闪存的速度快到足以在每个时钟周期提供一条指令。不过，当CPU时钟速率较高时，闪存的速度便无法跟上，因此系统必须引入等待状态，直至数据已就绪。

使用的等待状态数由预取模块控制寄存器（PRECON）的PFMWS位定义。默认情况下，PFMWS设置为七个等待状态。

1. 接下来，我们使用此默认值评估程序运行所需的时间。在user.c中编辑函数InitApp，确保按照以下所示在注释“// TODO:Set PREFEN mode”和“// TODO:Set PFMWS for wait states”处初始化PRECON

PRECONbits.PREFEN = 0;

PRECONbits.PFMWS = 7;

1. 编译、下载和运行程序。完成下表，确定各个循环迭代使用的CPU周期数。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器 时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 4059 | 15.86 | 63.42 |

1. 每个迭代的这种CPU周期数与您的预期值（接近六个周期）相比，结果如何？您认为二者相差如此之大的原因是什么？

63.42个周期/迭代时间远大于预期，很可能是因为对各个取指令操作使用了七个等待状态。

1. 现在我们将等待状态数减至最小值。对于66 MHz到133 MHz之间的速度，需要一个等待状态。对于133 MHz到200 MHz之间的速度，需要两个等待状态。PIC32MZ数据手册中指定了这些值。编辑函数InitApp，以便按照如下方式初始化PRECON：

PRECONbits.PREFEN = 0;

PRECONbits.PFMWS = 2;

1. 编译、下载和运行程序。完成下表，确定各个循环迭代使用的CPU周期数。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 3372 | 13.17 | 52.69 |

1. 现在，程序的运行速度变快了多少？您认为速度足够快吗？

现在，程序的每个迭代使用52.69个周期 – 将闪存等待状态从7降至2，进而将每个迭代的周期计数减少了约11个周期。与我们的预期相比，此周期计数仍然很大。

# 使用预取模块

预取模块的工作原理是：从闪存中一次性获取128位数据并将其存储到内部，从而为CPU提供零延时访问。有四个128位宽的独立行，一个用于CPU指令、一个用于CPU数据，其余两个用于外设访问。PRECON中的PREFEN字段用于控制预取模块的工作模式。

## CPU指令访问

1. 修改代码以将PREFEN设置为1。这将使预取模块对CPU请求的指令进行取指操作。
2. 编译、下载和运行程序。完成下表，确定各个循环迭代使用的CPU周期数。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 2936 | 11.47 | 45.88 |

## CPU指令和数据访问

1. 修改代码以将PREFEN设置为2。这将使预取模块对CPU请求的指令和数据进行取指操作。
2. 编译、下载和运行程序。完成下表，确定各个循环迭代使用的CPU周期数。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 2934 | 11.46 | 45.84 |

## 所有存储器访问

1. 修改代码以将PREFEN设置为3。这将使预取模块对CPU和外设请求的所有内容进行取指操作。
2. 编译、下载和运行程序。完成下表，确定各个循环迭代使用的CPU周期数。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 2936 | 11.47 | 45.88 |

1. 哪种工作模式带来了最大的改善？为什么您会这样认为呢？

针对CPU指令使用预取模块带来了最大的改善。其他模式的作用不大。原因是，与数据相比，程序必须访问更多的指令。

# 使用高速缓存

## 使能和清除高速缓存

1. 通过编辑文件pic32\_init\_cache.S（临近注释“// TODO:Select cache mode”，第60行附近）使能高速缓存。将代码更改为：

// TODO:Select cache mode

// #define \_\_PIC32\_CACHE\_MODE \_CACHE\_DISABLE

#define \_\_PIC32\_CACHE\_MODE \_CACHE\_WRITETHROUGH\_WRITEALLOCATE

// #define \_\_PIC32\_CACHE\_MODE \_CACHE\_WRITEBACK\_WRITEALLOCATE

请注意，在运行测试前，我们要确保高速缓存中不存在有用的数据，否则会影响我们的测试结果。我们让程序运行函数Pollute\_Cache，以访问大量其他数据，进而将数组aa从高速缓存中移出。将该代码添加到注释“// TODO: pollute cache”位置处。

Pollute\_Cache();

1. 编译、下载和运行程序。完成下表，确定各个循环迭代使用的CPU周期数。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 472 | 1.84 | 7.38 |

1. 每个迭代需要多少个CPU周期？将该值与至少7个周期的预期值相比，结果如何？为什么不是7个周期？

该值略大于7个周期，因为存在两个嵌套循环。我们预计内循环使用7个周期。不过，我们未考虑外循环的时序开销，此开销将使用另外的7个周期（分布在内循环的16个迭代中）。  
7/16 = 0.44，此结果应接近于观察到的差值。

## 更改数组大小

数据高速缓存最多可存储4 kB数据。我们已评估了使用16行 \* 16列 \* 4字节/元素 = 1024字节的数组的程序性能。该数组非常小，足以完全装入高速缓存。我们评估一下将数组大小增至4 kB或更大时会发生什么。

1. 相应地修改NUM\_ROWS和NUM\_COLS，以完成下面的表格。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 472 | 1.84 | 7.38 |
| **24** | **24** | 576 | 2304 | 1072 | 1.86 | 7.44 |
| **32** | **32** | 1024 | 4096 | 2147 | 2.10 | 8.39 |
| **48** | **48** | 2304 | 9216 | 4948 | 2.15 | 8.59 |
| **64** | **64** | 4096 | 16384 | 8801 | 2.15 | 8.59 |
| **96** | **96** | 9216 | 36864 | 19729 | 2.14 | 8.56 |
| **128** | **128** | 16384 | 65536 | 35009 | 2.14 | 8.55 |

1. 在数组大小接近或超出高速缓存大小时，每个迭代的CPU周期数如何变化？请相应说明。

对于能完全装入高速缓存中的16x16和24x24数组，每个迭代的周期数开始接近于预期值7。带32x32个元素或更多元素的数组不能完全装入高速缓存中，平均执行周期计数会增加约一个周期。

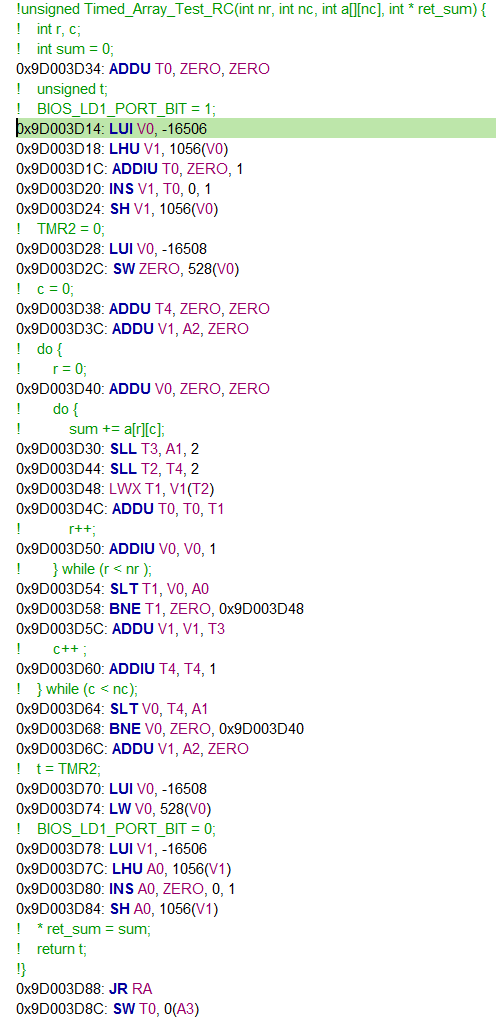
## 更改数组访问模式

C语言中的多维数组在存储器中的排列方式为：最后一维的元素在存储器是连续的。这种排列适用于高速缓存，因为一个高速缓存块可保存多个连续的元素。

我们来看一下，如果我们不访问按顺序存储在存储器中的元素，程序速度有何变化。

1. 更改程序以调用函数Timed\_Array\_Test\_**RC**（而非Timed\_Array\_Test\_**CR**）。
2. 通过启动调试器来编译程序并将其下载到MCU中。使用反汇编程序查看Timed\_Array\_Test\_RC内循环的目标代码。内循环中有多少个指令，您预计每个迭代将使用多少个周期？

内循环有六条指令（从地址0x9D003D48到0x9D003D6C），与Timed\_Array\_Test\_CR循环类似，因此，应使用相同的时间量（至少七个周期）。



1. 运行该代码以完成下表，进而根据需要修改NUM\_ROWS。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NUM\_ROWS** | **NUM\_COLS** | **数组大小（元素数）** | **数组大小（字节数）** | **定时器 时标** | **定时器 时标/迭代** | **CPU周期/迭代** |
| **16** | **16** | 256 | 1024 | 473 | 1.85 | 7.39 |
| **24** | **24** | 576 | 2304 | 1058 | 1.84 | 7.35 |
| **32** | **32** | 1024 | 4096 | 2141 | 2.09 | 8.36 |
| **48** | **48** | 2304 | 9216 | 5006 | 2.17 | 8.69 |
| **64** | **64** | 4096 | 16384 | 13407 | 3.27 | 13.09 |
| **96** | **96** | 9216 | 36864 | 30095 | 3.27 | 13.06 |
| **128** | **128** | 16384 | 65536 | 53439 | 3.26 | 13.05 |

1. 按行（而非按列）访问时，每个迭代的CPU周期数如何变化？

如果数据集大于高速缓存大小，则性能会变差。对于能完全装入高速缓存中的16x16和24x24数组，每个迭代的周期数开始接近于预期值7。32x32和48x48数组不能完全装入高速缓存中，平均执行周期计数会增加约一个周期。64x64以及更大的数组会使性能变得更差，因为每个循环迭代都增加了约六个周期。

1. 您如何解释这一点？

按行访问数据会消除空间局部性，这种局部性有助于之前示例中的高速缓存（按列访问）。每次访问都会导致一次高速缓存未命中，进而强制CPU在访问SRAM时等待4到5个周期。