TỐI ƯU CHƯƠNG TRÌNH C TRÊN VI XỬ LÝ ARM

Footprint & Speed

- Memory footprint: lượng bộ nhớ cần thiết để chạy chương trình.

- Tốc độ thực thi:
  - Đo bằng số cycle
  - Đo bằng đồng hồ

```
t = clock()
//
//      Program
//
t = (clock() – t) / CLOCKS_PER_SEC
```

Mục tiêu

- Chương trình khi chạy trên thiết bị cấu hình yếu cần:
  - Thỏa yêu cầu tiết kiệm bộ nhớ (đo bằng memory footprint)
  - Thỏa yêu cầu về tốc độ thực thi

→ Phát sinh vấn đề tối ưu khi lập trình

  - Giảm footprint
  - Tăng tốc độ

- Tối ưu mã Assembly
  - Phụ thuộc thiết bị
  - Độ phức tạp cao

- Tối ưu mã C

Nội dung

1. Tối ưu cho một hàm
   - Kiểu dữ liệu
   - Tham số hàm
   - Số lượng biến
2. Pipeline
   - Cấu trúc rẽ nhánh
3. Tối ưu vòng lặp
   - Số vòng lặp
   - Kích thước vòng lặp
4. Vấn đề bộ nhớ
   - Structure
   - Pointer Alias
   - Truy suất dữ liệu mảng
5. Phép chia
   - Số có dấu & không dấu
   - Phép chia tổng quát

Kiểu dữ liệu

- Các thao tác tính toán của MCU đều thực hiện trên các thanh ghi có kích thước cố định.
- Vd LPC 2378: 32 bit
- Đối với các biến <> 32 bit, Compiler phải phát sinh toán tử ép kiểu (Casting).

Ép kiểu

```
short add(short a, short b)
{
        return a + b;                   //implicit casting
}

short add(short a, short b)
{
        return (short)(a + b);          //explicit casting
}
```

```
MOV     r0, r0, LSL #16         ;r0 = (int)a
MOV     r1, r1, LSL #16         ;r1 = (int)b
ADD     r1, r1, r0, ASR #16     ;r1 = r1 + r0
MOV     r0, r1, LSL #16
MOV     r0, r1, ASR #16         ;r0 = (short)r1
MOV     pc, lr                  ;return r0
```

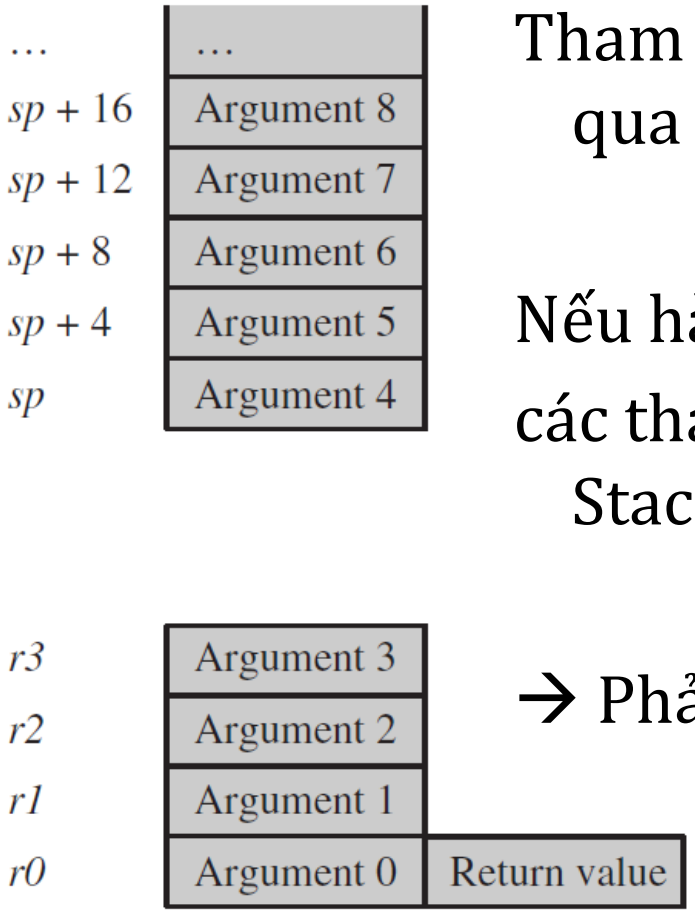
Không ép kiểu

```
int add(int a, int b)
{
        return a + b;
}
```

```
ADD     r0, r0, r1      ;r0 = r0 + r1
MOV     pc, lr          ;return r0
```

→ Nên sử dụng biến tương ứng kích thước thanh ghi.

Tham số hàm



Tham số hàm được truyền mặc định qua 4 thanh ghi r0, r1, r2, r3.

Nếu hàm có nhiều hơn 4 tham số,

các tham số còn lại sẽ được đặt trong Stack Pointer

→ Phải truy xuất SP để lấy dữ liệu

Số lượng biến

Các C Compiler trên ARM sử dụng 15 thanh ghi.

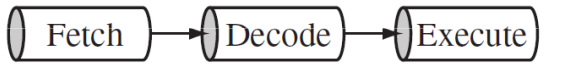
Trong đó r0 → r11 phục vụ cho các biến đa dụng.

→ Tránh việc nạp đi nạp lại các biến:

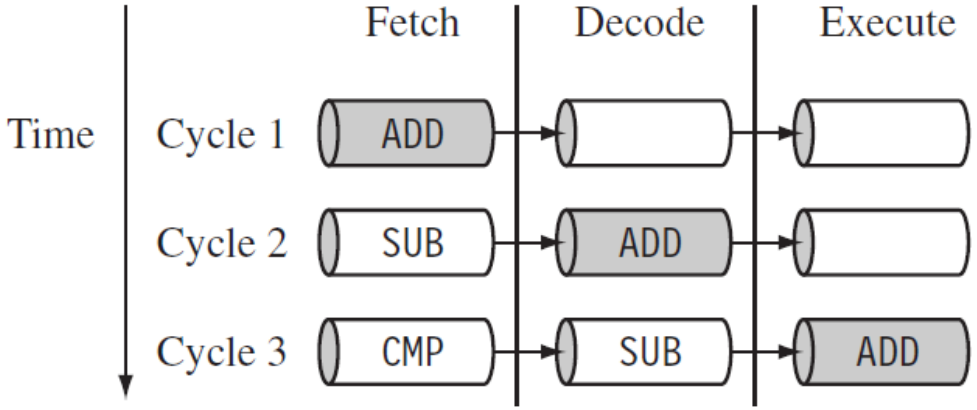
- → Số lượng biến của hàm nên trong phạm vi số lượng thanh ghi cho phép.
- → Đảm bảo các biến truy suất thường xuyên phải luôn nằm trên thanh ghi.
  - → Từ khóa *Register*
    - → Vấn đề Portability: Biến *Register* phụ thuộc Compiler.

Pipeline

- Lệnh được nạp và thực thi tuần tự theo cấu trúc của Pipeline.
- Cơ chế một lệnh được nạp lên khi đang thực hiện một lệnh khác.



ARM7 Three-stage pipeline.



Pipeline

- Các lệnh sau khi Execute có thể mất hơn 1 Cycle mới trả kết quả.
  - ARM7TDMI :
    - ALU: 1 Cycle
    - MUL: 5 Cycle

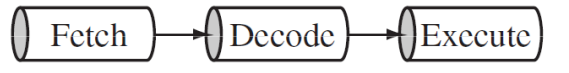
- Nếu lệnh n+1 phụ thuộc lệnh n:
  - Khi n đã Execute: n + 1 được nạp vào Pipeline nhưng vẫn phải đợi n thực hiện xong.
  - Khi n thực hiện xong: Khi đó Pipeline sẽ bị xóa sạch và lệnh n + 1 được nạp lại từ đầu → Vỡ Pipeline.

Pipeline

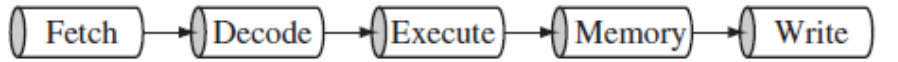
- Trên PC các CPU có BPU (Branch Prediction Unit).
- BPU: Sắp xếp các lệnh sao cho tối ưu Pipeline.
- MCU của ARM không có!

→ Tối ưu Pipeline bằng Code

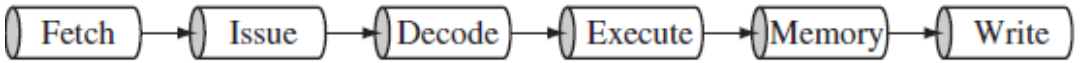
→ Vấn đề Portablility: MCU khác nhau có cấu trúc Pipeline khác nhau.



ARM7 Three-stage pipeline.



ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

→ Pipeline dài: Thực hiện đồng thời nhiều lệnh.

Pipeline và cấu trúc rẽ nhánh

Loop:

1. ADD r1, r1, #1 ; r1++
2. CMP r1, #0x40 ; compare r1, 64
3. BCC Loop ; if (r1<64) ***goto*** Loop
4. MOV pc, r14 ; return sum

- Nếu r1 < 64: thực hiện lại lệnh 1, trong khi lệnh 4 đã được nạp vào Pipeline.

→ Hạn chế rẽ nhánh:

→ Thay rẽ nhánh bằng chỉ mục.

→ Đặt các nhánh có xác suất cao ngay sau lệnh if.

Thay rẽ nhánh bằng chỉ mục

- Rẽ nhánh:

```
if (x == 0)
    color = red;
else
    color = green;
```

```
if (x < 0)
   y++;
```

- Không rẽ nhánh

```
arr = {red, green};
color = arr[x != 0];
```

- `y += x < 0;`

Lựa chọn nhánh có xác suất cao

```
dtb = (mon1 + mon2)/2;
if (dtb > 9)
    printf("Exellence");
else
    printf("Good");
```

```
if (rand() % 5 < 4)
    printf("0123");
else
    printf("4");
```

Tối ưu vòng lặp

- Mỗi lặp là 1 lần vỡ Pipeline

→ Giảm số vòng lặp (Unroll).

```
do
{
   s += i--;
}
while (i!= 0);
```

```
do
{
   s += i--;
   s += i--;
   s += i--;
   s += i--;
}
while (i!= 0);
```

- Giảm số vòng lặp xuống bao nhiêu là tốt?

Tối ưu vòng lặp

- Kích thước vòng lặp: số lệnh bên trong vòng lặp.
- Giảm số vòng lặp n lần → Kích thước code tăng n lần.

Ví dụ

- Chi phí vỡ Pipeline: 3 cycle
- Số instruction lặp lại: 65536
- Số vòng lặp: k
- Code size: s = 65536 / k * 4
- Chi phí loop: t = k * 3

Tối ưu vòng lặp

Số vònglặp	% Vòng chi phí lặp giảm xuống	Số lần Code tăng lên
65536	100	1
32768	50	2
16384	25	4
8192	12.5	8
4096	6.25	16
2048	3.125	32
1024	1.5625	64
512	0.78125	128
256	0.390625	256
128	0.195313	512
64	0.097656	1024
32	0.048828	2048
16	0.024414	4096
8	0.012207	8192
4	0.006104	16384
2	0.003052	32768
1	0.001526	65536

Tối ưu vòng lặp

- Nếu Unroll:
  - Làm tăng nhiều code size
  - Chi phí lặp giảm không đáng kể

→ Không nên unroll.

- Kích thước code size quá lớn:
  - Không đủ Cache để nạp → Tốn chi phí nạp code
  - Chương trình chậm đi rất nhiều.
  - Có khi không biên dịch được.

→ Giảm số vòng lặp phải bảo đảm giới hạn Code size.

Cấu trúc

```
struct {
  char a;
  int b;
  char c;
  short d;
}
```

Address	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

No padding:12 bytes

```
struct {
  char a;
  char c;
  short d;
  int b;
}
```

Address	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

Padding: 8 bytes

Cấu trúc

- Compiler truy suất struct tốt nhất trong phạm vi kích thước 128 bytes.
- Đối với các struct lớn nên chia nhỏ thành các struct con.

→ Tổng quát:

- → Xếp các biến có kích thước từ nhỏ đến lớn.
- → Xếp các mảng kích thước lớn sau cùng.
- → Đối với các struct lớn nên chia nhỏ thành các struct con.

Pointer Alias

```
void timers_v1(int *timer1, int *timer2, int *step)
{
    *timer1 += *step;
    *timer2 += *step;
}
```

```
LDR r3,[r0,#0]      ; r3 = *timer1
LDR r12,[r2,#0]     ; r12 = *step
ADD r3,r3,r12       ; r3 += r12
STR r3,[r0,#0]      ; *timer1 = r3
LDR r0,[r1,#0]      ; r0 = *timer2
LDR r2,[r2,#0]      ; r2 = *step
ADD r0,r0,r2        ; r0 += r2
STR r0,[r1,#0]      ; *timer2 = t0
MOV pc,r14          ; return
```

Compiler không nhận biết các Pointer có trỏ cùng một địa chỉ?

Nên không đảm bảo việc thay đổi ***timer1*** có ảnh hưởng đến ***step***?

→ ***step*** được load 2 lần.

Pointer Alias

```
void timers_v1(int *timer1, int *timer2, int *step)
{
    int n = *step;
    *timer1 += n;
    *timer2 += n;
}
```

```
LDR r12,[r3,#0]   ; r12 = *step
LDR r3,[r0,#0]    ; r3 = *timer1
ADD r3,r3,r12     ; r3 += r12
STR r3,[r0,#0]    ; *timer1 = r3
LDR r0,[r1,#0]    ; r0 = *timer2
ADD r0,r0,r12     ; r0 += r12
STR r0,[r1,#0]    ; *timer2 = t0
MOV pc,r14        ; return
```

→ Tránh sử dụng pointer để truy suất một địa chỉ không đổi nhiều lần.

Truy xuất dữ liệu mảng

int a[1024];

- Truy suất ngẫu nhiên

```
for (i = 0; i < 1204; i++)
    s += a[i];
```

```
LDR r1, [r0, r0, LSL #2]; r1=a[i]
ADD r2, r2, r1            ; s+=a[i]
```

→ Tốn chi phí cho phép dịch bit

- Truy suất tuần tự

```
int *p = a;
for (i = 0; i < 1024; i++)
    s += *p++;
```

```
LDR r1, [r0], #4          ; r1=*p++
ADD r2, r2, r1            ; s+=a[i]
```

Phép chia bằng dịch bit

- Số có dấu

```
int average_v1(int a, int b)
{
   return (a+b)/2;
}
```

```
ADD r0,r0,r1                ;r0 = a + b
ADD r0,r0,r0,LSR #31        ;if (r0<0) r0++
MOV r0,r0,ASR #1            ;r0 = r0 >> 1
MOV pc,r14                  ;return r0
```

1101 >> 1 = 1110

-3 >>1 = -2

-3 / 2 = -1

Phép chia bằng dịch bit

- Số không dấu

```
int average_v2(unsigned int a, unsigned int b)
{
   return (a+b)/2;
}
```

Compiler:

```
ADD r0,r0,r1                ;r0 = a + b
MOV r0,r0,ASR #1            ;r0 = r0 >> 1
MOV pc,r14                  ;return r0
```

→ Sử dụng số không dấu để đạt tốc độ cao hơn khi thực hiện phép chia.

Phép chia tổng quát

- ARM không tích hợp lệnh chia trên hardware. Các phép chia được thực hiện bằng các vòng lặp.
- Một phép chia số nguyên thường tốn 20 – 100 Cycles.

```
offset = (offset + increment) % buffer_size;
//Có thể tốn đến 50 Cycle.

offset += increment;
if (offset>=buffer_size)
   offset -= buffer_size;
//3 Cycle
```

Chuyển phép chia thành phép nhân

- Chia cho một số không đổi & thực hiện nhiều lần
- Giả sử cần tính q = n / d:

$s = (2^{32} - 1) / d + t$ $\quad (0 <= t < d)$

$\rightarrow s = 2^{32} / d - e$ $\quad e = (1 + t) / d.$

$q = n * s / 2^{32} - f$ $\quad (0 <= f <= 1)$

$= n /d - g$ $\quad g = e * n / 2^{32} + f$

$< e + f < 2$

$\rightarrow q = \{n/d, n /d - 1\}$

Chuyển phép chia thành phép nhân

```
void scale(unsigned int *dest,  unsigned int *src,  unsigned int d,  unsigned
    int N)  {
    unsigned int s = 0xFFFFFFFFu / d;
     unsigned int n, q, r;
    do
    {
         n = *src++;
        q = (unsigned int)(((unsigned long long)n * s) >> 32);
        r = n - q * d;
        q+= r>=d;
        *dest++ = q;
    }
    while (--N);
}
```

Tóm tắt

- Sử dụng biến 32 bit nếu không yêu cầu về không gian.
- Giới hạn tham số hàm và số biến cục bộ.
- Xếp cấu trúc theo tiêu chí tiết kiệm bộ nhớ.
- Giảm vòng lặp nhỏ nhưng thường xuyên.
- Truy suất dữ liệu mảng tuần tự khi có thể.
- Tránh sử dụng phép chia và số thực.
- Sử dụng số không dấu.

Tổng kết

- Time & Code:
  - Optimize mất nhiều thời gian để viết code hơn.
  - Optimize làm giảm tính dễ đọc, dễ điều chỉnh của code.
  - Lựa chọn giữa kích thước chương trình và tốc độ.
- Portability:
  - Optimize phụ thuộc compiler
  - Optimize phụ thuộc thiết bị

→ Optimize chỉ nên áp dụng khi cần thiết.