2025.11.8

정수 나눗셈은 사실 정수 곱셈

몰루

Kang MS

가져온 이유

왜 이런 걸 하는가

뭔가 말할만한 게 없나

찾아보다가 이거 왠지 재밌을 것 '같아서' 가져옴

해보고 실제로 재미있으면 더 하고

아니면 다른 주제 생각해봐야 함

발단

관찰해보자

관찰

observe

```
1  // divxy.c
2  int main(void) {
3      unsigned int x, y, z;
4      z = x/y;
5      return 0;
6  }
7
```

컴파일 해보자

gcc -S divxy.c

관찰

observe

```
1  // div
2  int ma
3      un
4      z
5      re
6  }
7
```

컴파일 해보자

gcc -S divxy.c

```
1   main:
2   .LFB0:
3       .cfi_startproc
4       endbr64
5       pushq   %rbp
6       .cfi_def_cfa_offset 16
7       .cfi_offset 6, -16
8       movq    %rsp, %rbp
9       .cfi_def_cfa_register 6
10      movl    -12(%rbp), %eax
11      movl    $0, %edx
12      divl    -8(%rbp)
13      movl    %eax, -4(%rbp)
14      movl    $0, %eax
15      popq    %rbp
16      .cfi_def_cfa 7, 8
17      ret
18      .cfi_endproc
```

관찰

observe

```
1  // divx3.c
2  int main(void) {
3      unsigned int x, z;
4      z = x/3;
5      return 0;
6  }
7
```

컴파일 해보자

gcc -S divx3.c

관찰

observe

```
1  // div
2  int ma
3      un
4      z
5      re
6  }
7
```

컴파일 해보자

gcc -S divx3.c

```
1   main:
2   .LFB0:
3       .cfi_startproc
4       endbr64
5       pushq   %rbp
6       .cfi_def_cfa_offset 16
7       .cfi_offset 6, -16
8       movq    %rsp, %rbp
9       .cfi_def_cfa_register 6
10      movl    -8(%rbp), %eax
11      movl    %eax, %edx
12      movl    $2863311531, %eax
13      imulq   %rdx, %rax
14      shrq    $32, %rax
15      shrl    %eax
16      movl    %eax, -4(%rbp)
17      movl    $0, %eax
18      popq    %rbp
19      .cfi_def_cfa 7, 8
20      ret
21      .cfi_endproc
```

관찰

observe

```
1   movl    -8(%rbp), %eax
2   movl    %eax, %edx
3   movl    $2863311531, %eax
4   imulq   %rdx, %rax
5   shrq    $32, %rax
6   shrl    %eax
```

컴파일 결과

왜 imulq 이고 왜 shrq 이지 나는 저런 거 한 적 없는데

모듈러 곱 역원 아님 여기서는 우연히 겹친 것

관찰

observe

저거 뭘까

저게 왜 될까

Proof

어디다가 쓸까

Application

저거 뭘까

저거 뭐야 왜 함

결과 해석

왜 저렇게 됐지

ACM SIGPLAN 의 PLDI

Programming Language Design and Implementation

ACM SIGPLAN: ACM Special Interest Group on Programming LANguage

Kang MS

결과 해석

왜 저렇게 됐지

PLDI 1994

Torbjorn Granlund, Peter L. Montgomery

Division by Invariant Integers Using Multiplication.

상수 정수는 곱셈으로 계산할 수 있다

4Byte 나눗셈 = 8B 곱셈 하나 8B right shift 하나와 동치

yubikey 만든 그룬란드와 동일인

몽고메리 알고리즘의 몽고메리와 동일인

dl.acm.org/doi/10.1145/178243.178249

결과 해석

왜 저렇게 하지

사칙연산 중에 가장 느린 연산

나눗셈. 나눗셈 > 곱셈 > 가감산

아예 지원 안 하는 ABI 도 있(었었)다.

RISC-V without M-extension, ATmega32 (sw)

근데 저게 곱셈으로 해결된다고?

광명

결과 해석

왜 저렇게 하지

Architecture/Implementation	N	Approx. Year	Time (cycles) for HIGH(N–bit $*$ N–bit)	Time (cycles) for N–bit/N–bit divide
Motorola MC68020 [18, pp. 9–22]	32	1985	41–44	76–78 (unsigned) 88–90 (signed)
Motorola MC68040	32	1991	20	44
Intel 386 [9]	32	1985	9–38	38
Intel 486 [10]	32	1989	13–42	40
Intel Pentium	32	1993	10	46
SPARC Cypress CY7C601	32	1989	40^{S}	100^{S}
SPARC Viking [20]	32	1992	5	19
HP PA 83 [16]	32	1985	45^{S}	70^{S}
HP PA 7000	32	1990	3^{FP}	70^{S}
MIPS R3000 [12]	32	1988	12^{P}	35^{P}
MIPS R4000 [17]	32	1991	12^{P}	75
	64		20^{P}	139
POWER/RIOS I [4, 22]	32	1989	5 (signed only)	19 (signed only)
PowerPC/MPC601 [19]	32	1993	5–10	36
DEC Alpha 21064AA [8]	64	1992	23^{P}	200^{S}
Motorola MC88100	32	1989	17^{S}	38
Motorola MC88110	32	1992	3^{P}	18

^{S} – No direct hardware support; approximate cycle count for software implementation
^{F} – Does not include time for moving data to/from floating point registers
^{P} – Pipelined implementation (i.e., independent instructions can execute simultaneously)

Table 1.1: Multiplication and division times on different CPUs

저게 왜 될까

증명해보자

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\left\lfloor \frac{x}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \leq x < 2^{32}$$

저 div 3 은 실수 나눗셈

Kang MS

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\left\lfloor \frac{x}{2^{33}} \cdot \frac{2^{33}}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \leq x < 2^{32}$$

저 div 3 은 실수 나눗셈

Kang MS

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\left\lfloor \frac{x}{2^{33}} \cdot \frac{2^{33}+1}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$

Kang MS

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\left\lfloor \frac{x}{3} + \frac{x}{3 \cdot 2^{33}} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$

$$\Rightarrow \frac{x}{3 \cdot 2^{33}} < \frac{1}{3}$$

Kang MS

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\left\lfloor \frac{x}{3} + \frac{x}{3 \cdot 2^{33}} \right\rfloor = \left\lfloor \frac{x}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$

$$\Rightarrow \frac{x}{3 \cdot 2^{33}} < \frac{1}{3}$$

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

작은 결론

$$\left\lfloor \frac{x}{2^{33}} \cdot \frac{2^{33}+1}{3} \right\rfloor = \left\lfloor \frac{x}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$

Kang MS

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\left\lfloor x \cdot \frac{2^{33}+1}{3} \cdot \frac{1}{2^{33}} \right\rfloor = \left\lfloor \frac{x}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$

Kang MS

Divide by 3

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\left\lfloor x \cdot \frac{2^{33}+1}{3} \right\rfloor \overset{u}{\gg} 33 = \left\lfloor \frac{x}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$

Kang MS

Divide by 3

$$M = \frac{2^{33}+1}{3}$$

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$\lfloor x \cdot M \rfloor \overset{u}{\gg} 33 = \left\lfloor \frac{x}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$
$$M \in \mathbb{N}, \quad 0 \le M < 2^{32}$$

Kang MS

Divide by 3

$$M = \frac{2^{33}+1}{3}$$

Unsigned 32bit 정수 x 를 3으로 나눠보자

$$(x \cdot M) \overset{u}{\gg} 33 = \left\lfloor \frac{x}{3} \right\rfloor$$

$$x \in \mathbb{N}, \quad 0 \le x < 2^{32}$$
$$M \in \mathbb{N}, \quad 0 \le M < 2^{32}$$

Kang MS

Divide by d

Unsigned 32bit 정수 x 의 나눗셈의 일반화

$$\left\lfloor \frac{x}{d} \right\rfloor = \left\lfloor \frac{x}{2^{32+s}} \cdot \frac{2^{32+s}}{d} \right\rfloor = \left\lfloor \frac{x}{2^{32+s}} \cdot \frac{2^{32+s}+k}{d} \right\rfloor$$

$$2^{s-1} < d < 2^s \quad k = \mathrm{rem}(-2^{32+s}, d)$$

$$\left\lfloor \frac{x}{2^{32+s}} \cdot \frac{2^{32+s}+k}{d} \right\rfloor = (x \cdot M) \overset{u}{\gg} (32+s)$$

$$M = \left\lceil 2^{32+s}/d \right\rceil < 2^{33}$$

Kang MS

Divide by d

Unsigned 32bit 정수 x 의 나눗셈의 일반화의 보정

$$\text{if } M > 2^{32},\ \text{use } M' = M - 2^{32} \text{ and}$$

$$\left\lfloor \left(\left(x \cdot M' \right) \overset{u}{\gg} 32 \right) + x \right\rfloor \overset{u}{\gg} s$$

$$\text{since } M < 2^{33},\ M' = M - 2^{32} < 2^{32}$$

$$\left\lfloor x \cdot M \cdot \frac{1}{2^{32+s}} \right\rfloor = \left\lfloor \frac{x}{d} \right\rfloor$$

/ 7 같은 경우는 보정이 들어감

어디다가 쓸까

application

In compiler

GCC & LLVM

GCC 와 LLVM 모두에 적용

다만 “얼마나”는 컴파일러에 따라 다름.

옛날에는 -fno-strength-reduce 로 꺼졌던 것 같은데 없어짐. 끄는 방법은 모르겠음

LLVM 같은 경우 -O3 옵션에서만 적용됨

In

GC

Table 1. Analysis which instruction is used for representing division on godbolt with different setups

Platform	-O3	Result
x86-64 gcc 10.2	no	imul, no div
x86-64 gcc 10.2	yes	imul, no div
x86-64 gcc 8.3	no	imul, no div
x86-64 gcc 8.3	yes	imul, no div
x86-64 clang 11.0.0	no	idiv
x86-64 clang 11.0.0	yes	imul
ARM gcc 9.2.1	no	umull, no div
ARM gcc 9.2.1	yes	umull, no div
ARM64 gcc 8.2	no	umull, no div
ARM64 gcc 8.2	yes	umull, no div
armv8-a clang 11.0.0	no	sdiv
armv8-a clang 11.0.0	yes	umull
RISC-V rv32gc clang 10.0.0	no	mulhu
RISC-V rv32gc clang 10.0.0	yes	mulhu
WebAssembly clang (trunk)	no	i32.div_s
WebAssembly clang (trunk)	yes	i32.div_u

reference

In code

Float division

정수 / 실수에서도 이게 통한다.

단, 출력값이 반드시 정수이고 실수 값이 1보다 클 때.

```
1  static const double INV_PI = 0.31830988618379067154;
2  static NOINLINE int mul_invpi_fn(int x){
3    return (int)(x * INV_PI);
4  }
```

In code

Float division

정수 / 실수에서도 이게 통한다.

단, 출력값이 반드시 정수이고 실수 값이 1보다 클 때.

```
1  static const uint64_t SRM_PI = 0xA2F9836Eu; // 2^33/pi
2  static NOINLINE int sr_divpi_fn(uint32_t x){
3    return (int)((x * SRM_PI) >> 33);
4  }
```

π = 3.14159265394106**46** 로 근사. 오차 1이 발생 가능

모든 unsigned int 32 에 대해 ±1 수준으로 작동. 최초 오차는 104,348에서 등장. (104348/33215)

Kang MS

In code

Float division

정수 / 실수에서도 이게 통한다.

단, 출력값이 반드시 정수이고 실수 값이 1보다 클 때.

```
1  static const uint64_t UNM_PI = 0xA2F9836E4Eu; // 2^41/pi
2  static NOINLINE int un_divpi_fn(uint32_t x){
3    return (int)((x * UNM_PI) >> 41); // ~5419350
4  }
```

π = 3.14159265359**0**9870 로 근사.

최초 오차가 5,419,351 에서 발생하는 대신, x 가 24bit 범위 넘어가는 순간 아예 다른 값이 나옴

Kang MS

In code

Float division

```
Int f(int x) { return x / M_PI; }
```

를 만든다고 하자. x의 범위는 상식적으로 낮은 값

```
return (int)(x * INV_PI);
```

부동소수점 상수 INV_PI 곱한 후 int 로 형변환

```
Return (int)((x * SRM_PI) >> 33)
```

Strength Reduction 적용.

33 값과 SRM_PI 값은 상황에 따라 조정 가능. 보통은 그냥 32 + log(dividor).

정말 엥간치 복잡한 상황이 아니면 후자가 더 빠름

In code

Float division

INV_PI 가 더 빠른 상황이 있을 수 있긴 함

Ex: 벡터화. 함수가 정말 순수해서 inline + 벡터화 당하면 INV_PI 가 더 빠름

But, 벡터화 안 되는 상황이면

실행시간 88% 감소 대략 1 / 0.12 = 8.4 배 향상.

그리고 저걸 회로로 만든다면?

후자가 비교 불가능하게 빠르고 만들기 편함

아니 fixed point 곱으로 회로 만들어도 되잖아요 → 후자가 정확히 그것

근데 또 당연히 이 방법이 벡터화가 불가능 한 것은 아님

In code

Float division

주의
원래 이정도로 간단한 함수는 저렇게
벤치하면 안 됩니다

INV_PI 가 더 빠른 상황이 있을 수 있긴 함

```
:~/assignment_test$ gcc bench_pi.c -o bench_pi && ./bench_pi
=== Divide-by-pi benchmark ===
N=0x800000, REPEAT=16
Divide 33554432 by pi:

identity:     33554432
double*invpi: 10680707
fixedpt SR:   10680707

identity     :    397.744 ms
double*invpi :    431.624 ms
fixedpt SR   :    401.670 ms
```

아니 fixed point 곱으로 회로 만들어도 되잖아요 → 후자가 정확히 그것

근데 또 당연히 이 방법이 벡터화가 불가능 한 것은 아님

기타등등

Others

Libdivide

"you can get a speedup of up to 10x for 64-bit integer division and a speedup of up to 5x for 32-bit integer division when using libdivide." (Supports SSE2, AVX2,AVX512)

V8 for chromiun

division-by-constant.h

리눅스 커널

reciprocal_div.h

런타임 나눗셈 최적화 용도로 사용. Cache 는 신이야

Torbjorn Granlund and Peter L. Montgomery. 1994. Division by invariant integers using multiplication.

Henry S. Warren, Jr. 2002. Hacker's Delight.

참고문헌

References

2025.11.8

질문받습니다

질문이 있다면 하십시오

Kang MS

확장판

Signed Operand

Signed value

부호 있는 값

당연히 부호 있는 값에서도 성립함

Ex: 벡터화. 함수가 inline 당하고 벡터화 당하면 INV_PI 가 더 빠름

증명도 비슷함

다만 다른 점은 음수의 나눗셈은 floor 가 아니라 celi 다. 올림으로 처리

비슷한 식 만들어놓고 특정 값이 -(1/d) 보다는 크다는 것을 증명

Hacker's delight 책에 같은 내용 증명까지 포함해서 적혀있음

Signed value

10–4 Signed Division by Divisors ≥ 2

At this point you may wonder if other divisors present other problems. We see in this section that they do not; the three examples given illustrate the only cases that arise (for $d \geq 2$).

Some of the proofs are a bit complicated, so to be cautious, the work is done in terms of a general word size W.

Given a word size $W \geq 3$ and a divisor d, $2 \leq d \leq 2^{W-1}$ we wish to find the least integer m and integer p such that

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{for } 0 \leq n < 2^{W-1}, \text{ and} \qquad (1a)$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \qquad \text{for } -2^{W-1} \leq n \leq -1, \qquad (1b)$$

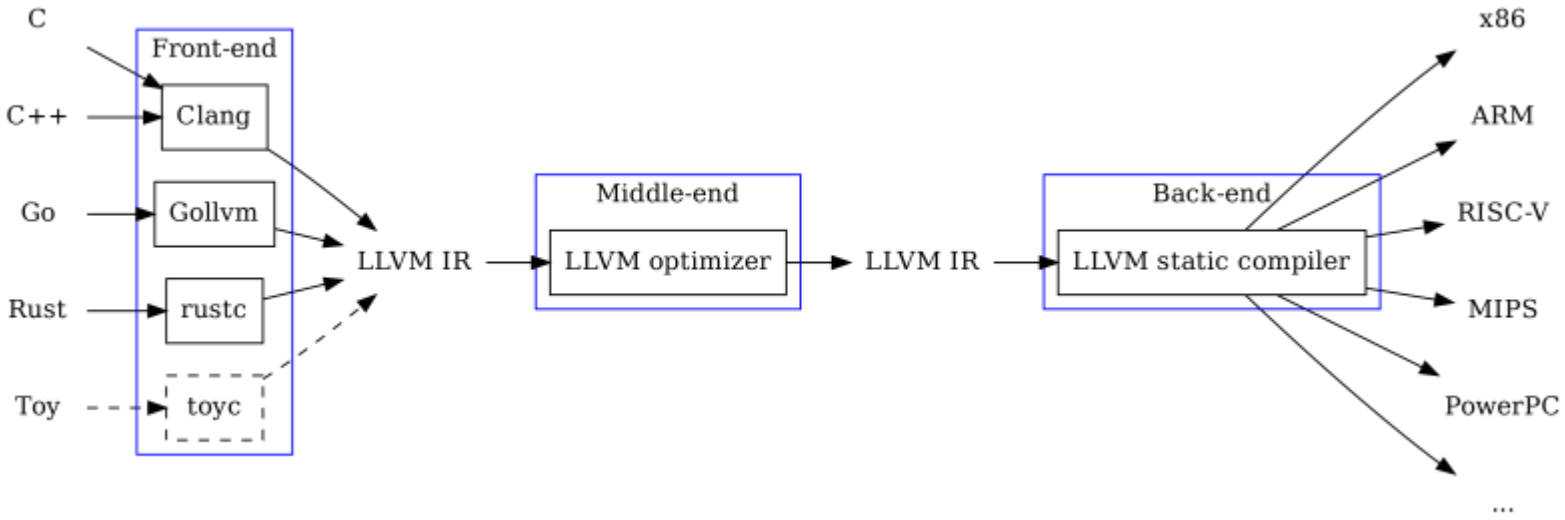
with $0 \leq m < 2^W$ and $p \geq W$.

Hacker's delight 책에 같은 내용 증명까지 포함해서 적혀있음

App: Compiler

컴파일러 상에서 적용 위치

컴파일 구조



언어 종속 / 최적화 / asm 제작

어떻게든 SSA를 만들고 그걸로 뭐라도 해보자

저 중간까지 가는 게 Byte code 같은 것들

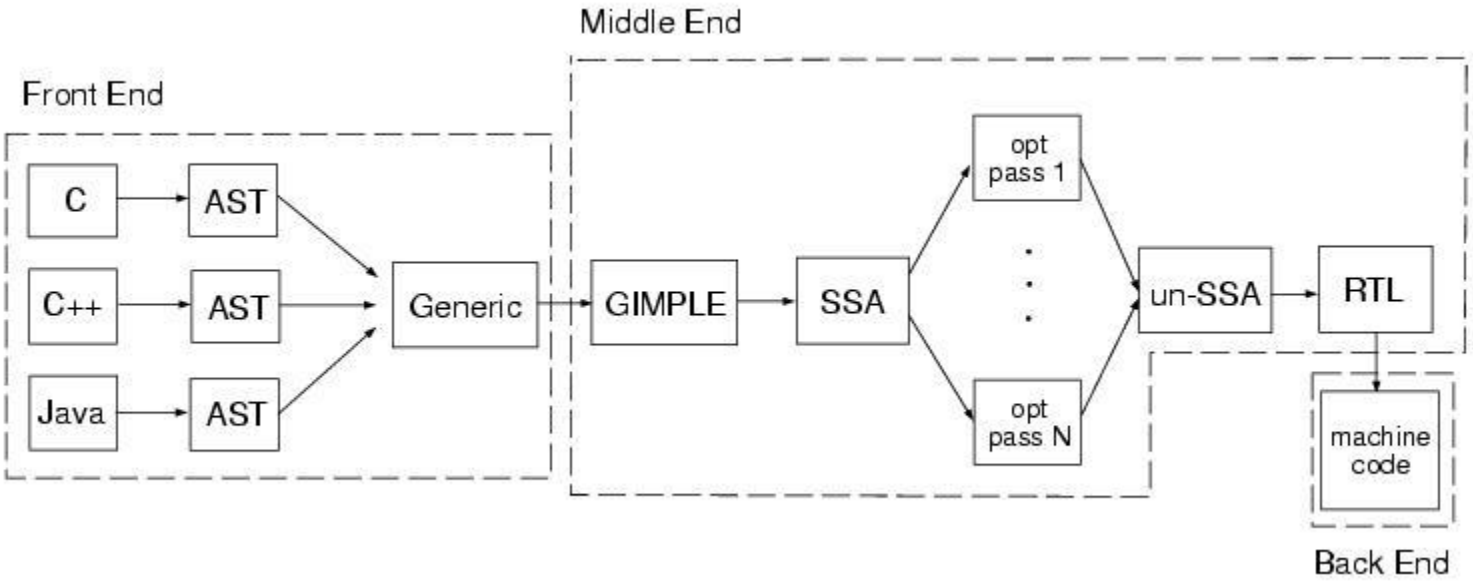
Image Source

Image Source

Text Source

컴파일 구조

Front end – Middle end – Back end



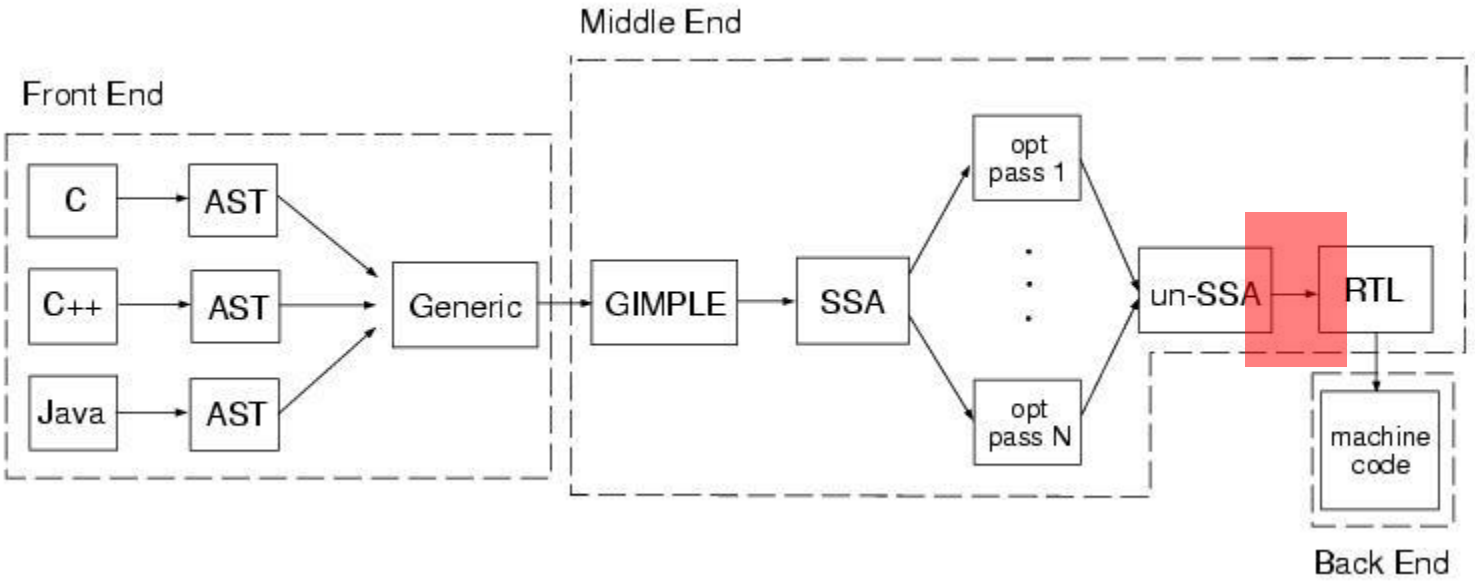
GCC: GIMPLE: G(eneric)+SIMPLE(IL)

GIMPLE is a three-address representation derived from GENERIC by breaking down GENERIC expressions into tuples of no more than 3 operands (with some exceptions like function calls). GIMPLE was heavily influenced by the SIMPLE IL used by the McCAT compiler project at McGill University, though we have made some different choices. For one thing, SIMPLE doesn't support goto.

Kang MS

컴파일 구조

Front end – Middle end – Back end



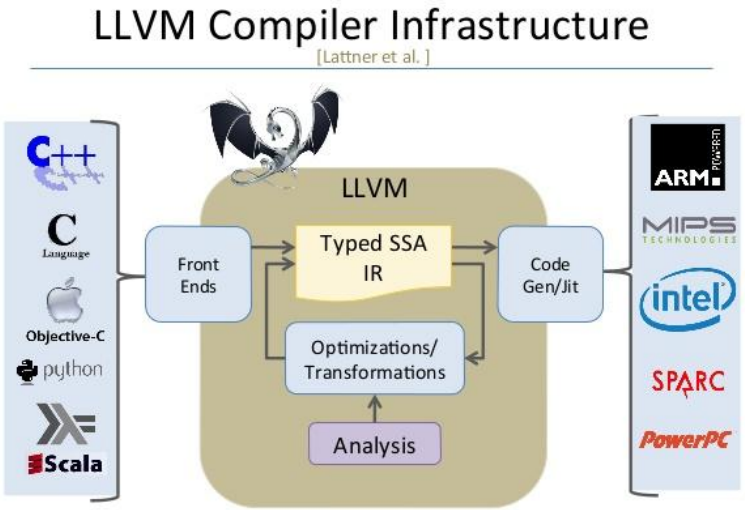
이 부분에서 하는 최적화다

RTL-expansion. Expmed.c

사실 최적화가 저기서 일어나기 쉽지 않음

컴파일 구조

Front end – Middle end – Back end



LLVM: LLVM IR

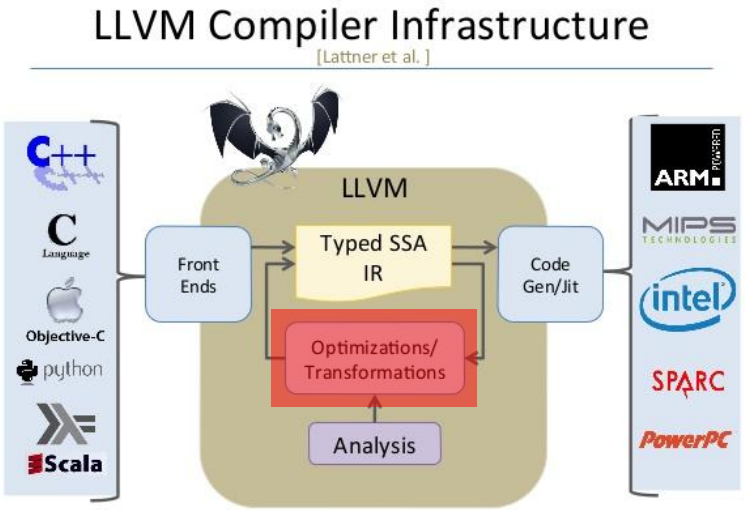
LLVM Intermediate Representation. 적외선 아님

Image Source

Kang MS

컴파일 구조

Front end – Middle end – Back end



이 부분에서 하는 최적화다

DivisionByConstantInfo.cpp L17~

Kang MS

TMI 0

뭔가 물어볼 것 같은 것

TMI 0

뭔가 물어볼 것 같은 것

저거 M 2^32의 곱 역원이랑 동치에요?

아님. 1 / 3 = 0 이지 3^{-1} 이 아니잖아요. 같은 값 나온 건 우연. 7 에선 no

단, 나눠지는 값이 d 으로 나누어 떨어진다는 것을 확신한다면 써도 됨

그러면 아예 순수 곱셈 하나라서 더 이점이 있겠죠

대신 Magic number 계산하기 복잡해짐: Extended Euclidean Algorithm

Hansel's Lamma

그리고 저런 상황이 딱히 많을 것 같지가 않음

저기서 우연이란? : $-2^{W+s} = 1 \pmod{d}$ 가 성립

Kang MS

TMI 1

Modular operation by invariant integer

TMI 1

Modular operation by invariant integer

“invariant 정수”의 나머지 연산도 가능함

By. Barrett reduction 이라는 다른 알고리즘. 그리고 x 가 적당히 작아야 함

E.g. to calculate the remainder of a division by 3 with a 32 bit multiply, multiply with 0x55555556 and extract the upper two bits; the result is exact for inputs up to 0x1fffffff. (gcc/expmed.c L3872~3874)

증명 생략

위키피디아에 잘 되어있더라

Kang MS