

Lab 4 – Intro to Assembly

Group 8 – Abuke, Gaerlan, Sy, Taruc

(Mishka + Tobey + Ethan) 1.

```
sall $2, %eax
```

- (shifts value in %eax left by 2 bits) $\rightarrow 2^2 = 4$
- shifting is much faster and simpler over multiplication for powers of two
- sall (shift all) shift entire section by a constant (2) number of bits

(Tobey + Mishka + Ethan) 2.

```
movl -4(%rbp), %edx    edx=x
movl %edx, %eax        eax=x
sall $2, %eax          eax = x << 2 = x * 4
addl %edx, %eax        eax = 4x + x = 5x
sall $3, %eax          eax = 5x << 3 = 5x * 8 = 40x
addl %edx, %eax        eax = 40x + x = 41x
```

- the code that's generated is
 - shift %eax by 2 bits ($x * (2^2) = 4x$) and store it back in %eax
 - add the input x (%edx) to %eax ($4x + x = 5x$)
 - shift %eax by 3 bits ($5x * (2^3) = 40x$)
 - add the input x (%edx) to %eax ($40x + x = 41x$)
 - $41x$ was decomposed to $((2^2x + x)(2^3) + x)$

(Tobey + Mishka + Ethan) 3.

```
movl -4(%rbp), %edx    edx = x
movl %edx, %eax        eax = x
sall $6, %eax          eax = x << 6 = x * 64 = 64x
subl %edx, %eax        eax = 64x - x = 63x
```

- the code that's generated is
 - shift %eax by 6 bits ($x * (2^6) = x * 64 = 64x$) and store it back in %eax
 - subtract the input x (%edx) from %eax ($64x - x = 63x$)
 - Essentially, $63x = (2^6)x - x$

(Tobey + Mishka + Alinus + Ethan) 4.

```
movl -4(%rbp), %edx    edx = x
movl %edx, %eax        eax = x
sall $2, %eax          eax = x << 2 = x * 4 = 4x
addl %edx, %eax        eax = 4x + x = 5x
negl %eax              eax = 0 - 5x = -5x
```

- the code that's generated is essentially $2 * 5$ but negated by two's complement
 - shift %eax by 2 bits ($x * (2^2) = 4x$) and store it back in %eax
 - add the input x (%edx) to %eax ($x + 4x = 5x$)
 - negates long ([does two's complement](#)) so the result is $2 * -5$

(Tobey + Mishka + Alinus) 5.

```
imull    $61, %eax, %eax      eax = eax * 61
```

- Compiler just uses imull (integer multiply) directly instead of shifts and adds

The compiler [optimizes](#) using the instruction count; apparently, they use [heuristics](#) which determine if the cost of shifting/adding is lower than the cost of using imull directly.

Looking at the algorithm, gcc tries to factor a number into factors of $x * 2^n + 1$, and then factors that number recursively until they reach 1 or a number that is handled by a special case. The compiler also has multiple different cases that attempt to account for as many numbers as possible. This is done through the `synth_mult` function which is under the `expmed.cc` of the [gcc compiler](#).

```
2607 /* t == 1 can be done in zero cost. */
2608 if (t == 1)
2609 {
2610     alg_out->ops = 1;
2611     alg_out->cost.cost = 0;
2612     alg_out->cost.latency = 0;
2613     alg_out->op[0] = alg_m;
2614
2615     return;
2616 }
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3179
3180
3181
3182
3183
3184
3185
3186
3187
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3197
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3297
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3397
3398
3399
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3497
3498
3499
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3597
3598
3599
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3697
3698
3699
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3797
3798
3799
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3887
3888
3889
3890
3891
3892
3893
3894
3895
3895
3896
3897
3898
3899
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3987
3988
3989
3990
3991
3992
3993
3994
3995
3995
3996
3997
3998
3999
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4088
4089
4090
4091
4092
4093
4094
4095
4096
4096
4097
4098
4099
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4188
4189
4190
4191
4192
4193
4194
4195
4195
4196
4197
4198
4199
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4248
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4287
4288
4289
4290
4291
4292
4293
4294
4295
4295
4296
4297
4298
4299
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4348
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4388
4389
4390
4391
4392
4393
4394
4395
4395
4396
4397
4398
4399
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4448
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4489
4490
4491
4492
4493
4494
4495
4495
4496
4497
4498
4499
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4548
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4569
4570
4571
4572
4573
4574
4575
4576
457
```

```

3007     /* T ends with ...01 or ...011.  Multiply by (T - 1) and add T.  */
3008
3009     op_cost = add_cost (speed, mode);
3010     new_limit.cost = best_cost.cost - op_cost;
3011     new_limit.latency = best_cost.latency - op_cost;
3012     synth_mult (alg_in, t - 1, &new_limit, mode);
3013
3014     alg_in->cost.cost += op_cost;
3015     alg_in->cost.latency += op_cost;
3016     if (CHEAPER_MULT_COST (&alg_in->cost, &best_cost))
3017     {
3018         best_cost = alg_in->cost;
3019         std::swap (alg_in, best_alg);
3020         best_alg->log[best_alg->ops] = 0;
3021         best_alg->op[best_alg->ops] = alg_add_t_m2;
3022     }
3023 }
3024
3025     /* We may be able to calculate a * -7, a * -15, a * -31, etc
3026 quickly with a - a * n for some appropriate constant n.  */
3027     m = exact_log2 (-orig_t + 1);
3028     if (m ≥ 0 && m < maxm)
3029     {
3030         op_cost = add_cost (speed, mode) + shift_cost (speed, mode, m);
3031         /* If the target has a cheap shift-and-subtract insn use
3032          that in preference to a shift insn followed by a sub insn.
3033          Assume that the shift-and-sub is "atomic" with a latency
3034          equal to it's cost, otherwise assume that on superscalar
3035          hardware the shift may be executed concurrently with the
3036          earlier steps in the algorithm.  */
3037         if (shiftsub1_cost (speed, mode, m) ≤ op_cost)
3038         {
3039             op_cost = shiftsub1_cost (speed, mode, m);
3040             op_latency = op_cost;
3041         }
3042     else
3043         op_latency = add_cost (speed, mode);
3044
3045     new_limit.cost = best_cost.cost - op_cost;
3046     new_limit.latency = best_cost.latency - op_latency;
3047     synth_mult (alg_in, (unsigned HOST_WIDE_INT) (-orig_t + 1) >> m,
3048                  &new_limit, mode);

```

```

3141  /* Try shift-and-add (load effective address) instructions,
3142  | i.e. do a*3, a*5, a*9. */
3143  if ((t & 1) != 0)
3144  {
3145    do_alg_add_t2_m:
3146      q = t - 1;
3147      m = ctz_hwi (q);
3148      if (q && m < maxm)
3149    {
3150      op_cost = shiftadd_cost (speed, mode, m);
3151      new_limit.cost = best_cost.cost - op_cost;
3152      new_limit.latency = best_cost.latency - op_cost;
3153      synth_mult (alg_in, (t - 1) >> m, &new_limit, mode);
3154
3155      alg_in->cost.cost += op_cost;
3156      alg_in->cost.latency += op_cost;
3157      if (CHEAPER_MULT_COST (&alg_in->cost, &best_cost))
3158    {

```

And then finally, the main algorithm, which is too large to show here, focuses on recursing through possible $2^n + 1$ scenarios

```

3065  /* Look for factors of t of the form
3066  | t = q(2**m +- 1), 2 ≤ m ≤ floor(log2(t - 1)).
3067  | If we find such a factor, we can multiply by t using an algorithm that
3068  | multiplies by q, shift the result by m and add/subtract it to itself.
3069
3070  We search for large factors first and loop down, even if large factors
3071  are less probable than small; if we find a large factor we will find a
3072  good sequence quickly, and therefore be able to prune (by decreasing
3073  COST_LIMIT) the search. */
3074
3075  do_alg_addsub_factor:
3076  for (m = floor_log2 (t - 1); m ≥ 2; m--)
3077  {
3078    unsigned HOST_WIDE_INT d;
3079
3080    d = (HOST_WIDE_INT_1U << m) + 1;
3081    if (t % d == 0 && t > d && m < maxm
3082    && (!cache_hit || cache_alg == alg_add_factor))
3083    {
3084      op_cost = add_cost (speed, mode) + shift_cost (speed, mode, m);
3085      if (shiftadd_cost (speed, mode, m) ≤ op_cost)
3086        op_cost = shiftadd_cost (speed, mode, m);
3087
3088      op_latency = op_cost;

```

We can try manually if 61 has good factors of 2^n , 2^{n-1} , and 2^{n+1} . We can get 61 with $31 + 15 + 15$. Which are three shifts and subtracts added together for a total of 9. Other factors are around or take more operations than this. Since imul takes [around 3 cycles](#). It is faster to just use imul.

(Tobey + Alinus + Ethan) 6.

Lots of using the leal function which seems to follow the [format](#) of

leal displacement(base register, offset register, scalar multiplier)

where

*base register + (offset register * scalar multiplier) + displacement*

Which often compresses and performs the operations of these functions without dereferencing the addresses

Leal is being used as an arithmetic instruction rather than for memory access. It allows the compiler to [combine multiple operations like addition and multiplication](#) by constants into a single instruction, instead of using several separate shift and add instructions. leal is a faster way of doing addition + multiplication operations. Leal works best for shift and add operations (see screenshot) $2^n + 1$. So if the synth_mult stumbles upon a $2^n + 1$ number, it can use leal for it instead of multiple shifts and adds.

In addition to that, when using the -O flag, the compiler also removes unnecessary instructions such as extra moves between registers, and redundant computations. Some functions keep the same basic ALU instructions because they are already optimal, while others are rewritten to use leal since it can do the same work in fewer instructions.

The first case is now

```
leal    0(%rdi,%rdi,4), %eax
```

Which roughly translates to $0 + (a * 4) + 0 \rightarrow eax = a * 4$

The second case is now

```
leal    (%rdi,%rdi,4), %eax  
leal    (%rdi,%rax,8), %eax
```

Which translates to

$a + (a * 4) + 0 \rightarrow eax = a * 5$
 $a + (a * 5 * 8) + 0 \rightarrow eax = a * 41$
(rax is the 64 bit extended eax)

The third case is actually still the same.

The fourth case is now

```
leal    (%rdi,%rdi,4), %eax  
negl    %eax
```

Which translates to

$a + (a * 4) + 0 \rightarrow eax = a * 5$
 $0 - a * 5 \rightarrow eax = -a * 5$

The fifth case is still using imull

```
imull   $61, %edi, %eax
```