Cross Compiling e Análise de Código Assembly, Registradores e Pipeline – Arquitetura de Computadores

Emanuel Santana Santos

Curso de Engenharia de computação – Universidade Estadual de Feira de Santana – Campus Feira de Santana
Emanuel.santana1514@gmail.com

1. Introdução

Este relatório tem como objetivo demonstrar os resultados da Cross Compiling (compilação cruzada), que permite gerar código assembly para outras máquinas e ou dispositivos que utilizam processadores de arquiteturas diferentes. E tudo isso através de um único dispositivo. Com estes códigos de diferentes arquiteturas é possível analisar as diferenças entre os códigos assembly gerados para ARM e Intel, uma vez que os códigos gerados para o ARM utilizam mais instruções, porém instruções mais simples baseadas em Reduced Instruction Set Computer (RISC). Enquanto que Intel baseia-se em Complex Instruction Set Computer (CISC).

Além disso é necessário fazer análise do funcionamento do pipeline, analisando os registradores utilizados em algumas operações que são feitas pelos códigos C e que serão convertidos em assembly.

2. Sistemas Utilizados

Para a confecção do projeto foi solicitado que o sistema operacional utilizado fosse baseado em Linux, então foi utilizado o Linux Ubuntu versão 20.04.1. Este sistema Linux foi utilizado por meio do Virtual Box versão 6.1.12 que possibilitou a utilização sem necessariamente criar um dual boot ou instalação do sistema como principal na máquina utilizada.

Além do sistema operacional também foi solicitado o uso do compilador Gnu Compiler Collection (GCC), e este foi utilizado na versão GCC 9.3.0, que é a versão default ao fazer a instalação pelo apt install do sistema Linux. Com este compilador já é possível gerar código assembly para Intel e Amd, mas para gerar código para arquitetura ARM é necessário instalar um Cross Compiling também do GCC para AARCH64 também da versão 9.3. Além disso também é possível fazer compilação para um sistema de 32 bits instalando um compilador específico, com isso a opção default compilar para ARM v7, e então é adicionada a tag "-march = armv8-a" para compilar o código para uma versão ARM v8 de 32bits, sendo a parte após o '=' alterável para outras versões disponíveis no compilador, todas as informações podem ser consultadas por meio da documentação do próprio compilador.

Além dos sistemas utilizados acima foram utilizadas opções do compilador que ajudam a otimizar o código gerado assim como no trabalho anterior, abaixo uma melhor descrição de cada nível de otimização e o que cada uma faz no código.

• -O0: Não realiza nenhuma otimização e cada comando do código-fonte é convertido diretamente na instrução correspondente no arquivo executável.

• -O1: O compilador visa reduzir tamanho do código e o tempo de execução. Permitindo reordenação de instruções, otimizações nos laços de repetição, e quando faz isso elimina código morto ou desnecessário.

• -O2: Aprimora ainda mais que -O1, incluindo otimizações para agendamento de instruções, otimização de chamadas recursivas.

• -O3: Otimização que visa melhorar o tempo de execução da aplicação. Move condições invariantes do laço de repetição, função embutida. Código maior - Mais custoso.

Essas otimizações são feitas através de flags que são ativadas quando cada comando é utilizado na compilação, no nível 2 são utilizadas todas as flags utilizadas no -O0 e -O1 e adiciona outras flags que permitem otimizar ainda mais o código, isso aumenta o tempo de compilação e uso de memória para melhorar ainda mais o código final gerado. Isso também pode ser observado no nível -O3, que também reutiliza as flags dos níveis anteriores e adiciona algumas outras a fim de otimizar ainda mais o código.

Registradores

São 16 registradores de 32 bits dos registradores acessíveis, 13 são registradores de propósito geral (r0 a r12). Os outros quatro registradores têm funções específicas: Registradores R0 a R7 são os mesmos em todos os modos de CPU, eles nunca são acumulados.

Registradores R8 a R12 são os mesmos em todos os modos de CPU, exceto no modo FIQ. O modo FIQ possui seus próprios registros R8 a R12.

R13 e R14 são colocados em todos os modos de CPU privilegiados, exceto o modo de sistema. Ou seja, cada modo que pode ser inserido devido a uma exceção tem seu próprio R13 e R14. Esses registradores geralmente contêm o ponteiro da pilha e o endereço de retorno das chamadas de função, respectivamente.

Pseudônimo:

R13 também é conhecido como SP, o Stack Pointer (apontador de pilha).

R14 também é conhecido como LR, o Link Register (registro de links), Esse registrador recebe o endereço de retorno em chamadas de procedimento.

R15 também é referido como PC, o Program Counter (contador de programas), indica o endereço da próxima instrução a ser executada.

Pipeline

Para conseguir que, em regime, uma instrução seja executada a cada ciclo de relógio, a arquitetura ARM utiliza uma técnica de implementação conhecida como pipeline de execução. Nessa implementação, a execução de uma instrução é dividida em estágios, cada um responsável por uma tarefa independente.
O processo de buscar a próxima instrução enquanto a instrução atual está sendo executada é chamado de “pipelining”. O pipelining é suportado pelo processador para

aumentar a velocidade de execução do programa. Aumenta o rendimento. Várias operações ocorrem simultaneamente, em vez de serialmente no pipelining.

Considerando que o pipeline em questão tenha três estágios eles são os seguintes: **Buscar:** Neste estágio, o processador ARM busca a instrução da memória.

Decodificar: nesta etapa reconhece a instrução que deve ser executada.

Executar: Neste estágio, o processador processa a instrução e escreve o resultado de volta no registrador desejado.

Se esses três estágios de execução forem sobrepostos, alcançaremos maior velocidade de execução. Esse pipeline existe a partir da versão 7 do processador ARM. Depois que o pipeline é preenchido, cada instrução requer um ciclo para completar a execução.

Considerando que existam 3 instruções, no primeiro ciclo, o processador busca a instrução 1 da memória. No segundo ciclo, o processador busca a instrução 2 da memória e decodifica a instrução 1. No terceiro ciclo, o processador busca a instrução 3 da memória, decodifica a instrução 2 e executa a instrução 1. Em no quarto ciclo, o processador busca a instrução 4, decodifica a instrução 3 e executa a instrução 2. O pipeline, portanto, executa uma instrução em três ciclos, isto é, fornece uma taxa de transferência igual a uma instrução por ciclo.

A quantidade de trabalho realizado em cada estágio pode ser reduzida aumentando o número de estágios no pipeline. Para melhorar o desempenho, o processador pode ser operado em uma frequência de operação mais alta.

À medida que mais ciclos são necessários para preencher o pipeline, a latência do sistema também aumenta. A dependência de dados entre os estágios também pode ser aumentada conforme os estágios do pipeline aumentam. Portanto, as instruções precisam ser agendadas durante a gravação do código para diminuir a dependência de dados.

A organização de um processador ARM com pipeline de três estágios consiste no seguinte: Banco de registros: inclui vários registros, conforme visto no modelo de ARM do programador.

A ALU: esta unidade executa as várias operações aritméticas e lógicas

Registrador de endereço e incremento: O registro armazena o endereço e o incrementador incrementa o mesmo para apontar para a próxima instrução. . **Registro de dados**: é usado como um buffer para armazenar os dados quando gravados na memória ou lidos da memória.

Decodificador de instrução: Como o nome diz, ele decodifica a instrução e emite os sinais de controle de acordo.

3. Experimentos

Diferença do código Intel para o ARM, otimização, registradores, instruções, e diferença entre os códigos.

Código 7: O código 7 define um tamanho de uma variável para 1024, cria três vetores do tipo inteiro com esse tamanho máximo, cria uma variável inteira para ser usada no laço e logo após há um laço para fazer a soma dos valores guardados em cada posição dos vetores “X” e “Y” salvando essa soma no vetor “Z”.

- ARM – O código ARMv8a gerado para o código do exemplo 7, quando convertido para assembly no nível de otimização -O0 (Sem otimização) continha um total de 74 linhas, esse código gerado utiliza vários registradores de uso comuns e alguns especiais, no código foram utilizados do r0 até o r7 e os especiais foram: SP (Stack pointer)– Que aponta para a pilha, o PC (Progam Counter) - contador de programa e o LR (Link Register) – que é o registrador de ligação.

  O código gerado com o nível de otimização -O1 gerou apenas 5 linhas, fazendo uma cópia do valor 1024 para o registrador R3 com a instrução MOV, operações com SUBS, faz desvio com BNE para continuar o laço e então usa MOV para copiar o valor para 0 e BX juntamente com o lr para fazer o retorno.

  No nível -O2 existem apenas duas linhas de código assembly, uma instrução MOV que registra no R0 o imediato “0” e depois uma chamada de retorno com o comando BX lr, já que o endereço de retorno está ligado no registrador lr.

  No nível -O3, também há apenas duas linhas de comando assembly e o comando é igual ao comando existente no nível de otimização -O2.

- Intel – O código gerado para Intel no exemplo 7 continha um total de 36 linhas quando foi gerado o código assembly com o nível de otimização -O0 (Sem otimização), foi gerado o código completo para o código C inclusive instruções que não serão utilizadas pelo programa. Há também um detalhe que os registradores Intel não são tão fáceis de identificar como os registradores ARM, então fica um código menor, porém, um pouco mais complexo de ser entendido. Mas quando comparados com um código que tem a mesma função, porém utiliza tipos diferentes é possível fazer a identificação desses registradores.

  No nível de otimização -O1 o compilador gerou apenas 5 linhas de código, onde essas instruções são duas instruções MOV para atribuição de valor em um registrador, uma instrução de subtração SUB, uma instrução de salto JNE e por último uma instrução de retorno RET

  No nível de otimização -O2 o compilador gerou apenas 2 linhas, onde há apenas uma instrução XOR comparando o registrador ‘eax’ com ele mesmo, e por último uma instrução de retorno.

  No nível de otimização -O3 o compilador gerou apenas 2 linhas assim como no nível anterior, e as instruções utilizadas foram as mesmas que no nível anterior.

Código 8: O código 8 é muito parecido com o código 7, também define um tamanho máximo de uma variável, cria três vetores com esse tamanho máximo mas do tipo double, também cria uma variável inteira para ser usada no laço e logo após há um laço para fazer a soma dos valores guardados em cada posição dos vetores “X” e “Y” salvando essa soma no vetor “Z”, assim como no código 7.

- ARM – O código ARMv8a gerado para o código do exemplo 8, quando convertido para assembly no nível de otimização -O0 (Sem otimização) continha um total de 80 linhas, esse código gerado utiliza vários registradores de uso comuns e alguns especiais, no código, assim como no exemplo 7 também foram utilizados os registradores do r0 até o r7 e os especiais foram: SP – Que aponta para a pilha, o PC - contador de programa e o LR – que é o registrador de ligação.

  Já o código gerado com o nível de otimização -O1 gerou apenas 5 linhas, assim como o 7, tendo a mesma função, fazendo uma cópia do valor 1024 para o registrador R3 com a instrução MOV, operações com SUBS, faz desvio com BNE para continuar o laço e então usa MOV para copiar o valor para 0 e BX juntamente com o lr para fazer o retorno.

  No nível -O2 assim como o exemplo 7 existem apenas duas linhas de código assembly, uma instrução MOV que registra no R0 o imediato “0” e depois uma chamada de retorno com o comando BX lr, já que o endereço de retorno está ligado no registrador lr.

  No nível -O3, também há apenas duas linhas de comando assembly e o comando é igual ao comando existente no nível de otimização -O2.

- Intel – O código gerado para Intel no exemplo 8 continha um total de 35 linhas quando foi gerado o código assembly com o nível de otimização -O0 (Sem otimização), foi gerado o código completo para o código C inclusive instruções que não serão utilizadas pelo programa. Esse código utiliza alguns registradores diferentes do exemplo anterior, além disso algumas instruções são instruções que acionam flags, como este código apresenta variáveis do tipo double, esses registradores são utilizados para que sejam realizadas operações com esses registradores.

  No nível de otimização -O1 assim como no código anterior o compilador gerou apenas 5 linhas de código, onde essas instruções são duas instruções MOV para atribuição de valor em um registrador, uma instrução de subtração SUB, uma instrução de salto JNE e por último uma instrução de retorno RET, assim como o exemplo 7.

  No nível de otimização -O2 o compilador gerou apenas 2 linhas, onde há apenas uma instrução XOR comparando o registrador ‘eax’ com ele mesmo, e por último uma instrução de retorno.

  No nível de otimização -O3 o compilador gerou apenas 2 linhas assim como no nível anterior, e as instruções utilizadas foram as mesmas que no nível anterior.

Código 9: O código 9 também define um tamanho de 1024 cria três ponteiros do tipo inteiro e uma variável também inteira para ser utilizada no laço, utiliza esses ponteiros para alocação de memória para variáveis do tipo inteira e dentro do laço faz operações de soma utilizando o valor apontado na memória pelos ponteiros com a soma da variável "I" que refere-se ao valor atual do ciclo no laço e logo após essa soma, o resultado dela é atribuído em um dos ponteiros criados, nesse caso o ponteiro "Z".

- ARM - O código ARMv8a gerado para o código do exemplo 9, quando convertido para assembly no nível de otimização -O0 (Sem otimização) continha um total de 47 linhas, esse código gerado utiliza vários registradores de uso comuns e alguns especiais, no código, assim como nos também foram utilizados os registradores do r0 até o r7 e os especiais foram: SP – Que aponta para a pilha, o PC - contador de programa e o LR – que é o registrador de ligação, além disso há também uma instrução com alocação de memória .

  Já o código gerado com o nível de otimização -O1 gerou apenas 5 linhas, assim como os códigos anteriores, tendo a mesma função, fazendo uma cópia do valor 1024 para o registrador R3 com a instrução MOV, operações com SUBS, faz desvio com BNE para continuar o laço e então usa MOV para copiar o valor para 0 e BX juntamente com o lr para fazer o retorno.

  No nível -O2 assim como os exemplos anteriores existem apenas duas linhas de código assembly, uma instrução MOV que registra no R0 o imediato "0" e depois uma chamada de retorno com o comando BX lr, já que o endereço de retorno está ligado no registrador lr.

  No nível -O3, também há apenas duas linhas de comando assembly e o comando é igual ao comando existente no nível de otimização -O2.

- Intel – O código gerado para Intel no exemplo 9 continha um total de 41 linhas quando foi gerado o código assembly com o nível de otimização -O0 (Sem otimização), foi gerado o código completo para o código C inclusive instruções que não serão utilizadas pelo programa. Como esse código utiliza alocação de memória, há uma chamada para essa utilização, mas em relação aos códigos anteriores não há registradores especiais utilizados apenas por essa alocação de memória com o 'malloc'.

  No nível de otimização -O1 assim como no código anterior o compilador gerou apenas 5 linhas de código, onde essas instruções são duas instruções MOV para atribuição de valor em um registrador, uma instrução de subtração SUB, uma instrução de salto JNE e por último uma instrução de retorno RET, assim como o exemplo anterior.

  No nível de otimização -O2 o compilador gerou apenas 2 linhas, onde há apenas uma instrução XOR comparando o registrador 'eax' com ele mesmo, e por último uma instrução de retorno.

  No nível de otimização -O3 o compilador gerou apenas 2 linhas assim como no nível anterior, e as instruções utilizadas foram as mesmas que no nível anterior.

Código 10: O código 10, assim como o 9 também define um tamanho de 1024 cria três ponteiros porém eles são do tipo double e uma variável também inteira para ser utilizada no laço, utiliza esses ponteiros para alocação de memória e dentro do laço faz operações de soma utilizando o valor apontado na memória pelos ponteiros com a soma da variável "I" que refere-se ao valor atual do ciclo no laço e logo após essa soma, o resultado dela é atribuído em um dos ponteiros criados, nesse caso o ponteiro "Z" assim como é feito no código 9.

- ARM – O código ARMv8a gerado para o código do exemplo 10, quando convertido para assembly no nível de otimização -O0 (Sem otimização) também continha um total de 47 linhas, esse código gerado utiliza vários registradores de uso comuns e alguns especiais, no código, assim como nos também foram utilizados os registradores do r0 até o r7 e os especiais foram: SP – Que aponta para a pilha, o PC - contador de programa e o LR – que é o registrador de ligação, além disso há também uma instrução com alocação de memória .

  Já o código gerado com o nível de otimização -O1 gerou apenas 5 linhas, assim como os códigos anteriores, tendo a mesma função, fazendo uma cópia do valor 1024 para o registrador R3 com a instrução MOV, operações com SUBS, faz desvio com BNE para continuar o laço e então usa MOV para copiar o valor para 0 e BX juntamente com o lr para fazer o retorno.

  No nível -O2 assim como os exemplos anteriores existem apenas duas linhas de código assembly, uma instrução MOV que registra no R0 o imediato "0" e depois uma chamada de retorno com o comando BX lr, já que o endereço de retorno está ligado no registrador lr.

  No nível -O3, também há apenas duas linhas de comando assembly e o comando é igual ao comando existente no nível de otimização -O2.

- Intel – O código gerado para Intel no exemplo 10 continha um total de 41 linhas assim como o código 9 quando foi gerado o código assembly com o nível de otimização -O0 (Sem otimização), foi gerado o código completo para o código C inclusive instruções que também não serão utilizadas pelo programa. Em relação ao código anterior apenas algumas instruções de flags foram utilizadas e em relação ao anterior alguns registradores de ponto flutuante também foram utilizados ao invés dos registradores usados no código 9.

  No nível de otimização -O1 assim como no código anterior o compilador gerou apenas 5 linhas de código, onde essas instruções são duas instruções MOV para atribuição de valor em um registrador, uma instrução de subtração SUB, uma instrução de salto JNE e por último uma instrução de retorno RET, assim como o exemplo anterior.

  No nível de otimização -O2 o compilador gerou apenas 2 linhas, onde há apenas uma instrução XOR comparando o registrador 'eax' com ele mesmo, e por último uma instrução de retorno.

  No nível de otimização -O3 o compilador gerou apenas 2 linhas assim como no nível anterior, e as instruções utilizadas foram as mesmas que no nível anterior.

Código 11: O código 11 cria quatro variáveis inteiras e após isso faz duas condicionais comparando valores e após essa comparação faz uma atribuição em uma das variáveis criadas.

- ARM – O código ARMv8a gerado para o código do exemplo 11, quando convertido para assembly no nível de otimização -O0 (Sem otimização) continha um total de 40 linhas, esse código gerado utiliza vários registradores de uso comuns e alguns especiais, no código, assim como nos também foram utilizados os registradores do r0 até o r7 e os especiais foram: SP – Que aponta para a pilha, o PC - contador de programa e o LR – que é o registrador de ligação, além disso há também uma instrução com alocação de memória .

  Já o código gerado com o nível de otimização -O1 gerou apenas 2 linhas, enquanto os códigos anteriores geralmente geravam 5 linhas nesse nível de otimização, mas como esse código não realizava operações com laço e nem alocação de memória, pode existir alguma relação com esse tamanho de código. Nas duas linhas de código existentes usa MOV para copiar o valor para 0 no registrador e BX juntamente com o LR para fazer o retorno.

  No nível -O2 assim como os exemplos anteriores existem apenas duas linhas de código assembly, uma instrução MOV que registra no R0 o imediato “0” e depois uma chamada de retorno com o comando BX lr, já que o endereço de retorno está ligado no registrador lr.

  No nível -O3, também há apenas duas linhas de comando assembly e o comando é igual ao comando existente no nível de otimização -O2.

- Intel – O código gerado para Intel no exemplo 11 continha um total de 25 linhas quando foi gerado o código assembly com o nível de otimização -O0 (Sem otimização), foi gerado o código completo para o código C inclusive instruções que não serão utilizadas pelo programa.

  No nível de otimização -O1 o compilador gerou apenas 2 linhas, onde há apenas uma instrução MOV colocando o valor imediato 0 no registrador ‘eax’, e por último uma instrução de retorno.

  No nível de otimização -O2 há uma mudança, o compilador gerou apenas 2 linhas também assim como no nível anterior, porém trocou a instrução MOV po ruma XOR, que compara o registrador ‘eax’ com ele mesmo, e por último uma instrução de retorno.

  No nível de otimização -O3 o compilador gerou apenas 2 linhas assim como no nível anterior, e as instruções utilizadas foram as mesmas que no nível anterior.

Código 12: O código 12 também cria quatro variáveis, porém três do tipo double e uma do tipo inteira, após isso faz duas condicionais comparando valores e após essa comparação faz uma atribuição na variável inteira.

- ARM – O código ARMv8a gerado para o código do exemplo 11, quando convertido para assembly no nível de otimização -O0 (Sem otimização) continha um total de 47 linhas, esse código gerado utiliza vários registradores de uso comuns e alguns especiais, no código, assim como nos também foram utilizados os registradores do r0 até o r7 e os especiais foram: SP – Que aponta para a pilha, o PC - contador de programa e o LR – que é o registrador de ligação, além disso há também uma instrução com alocação de memória .

  Já o código gerado com o nível de otimização -O1 gerou apenas 2 linhas, enquanto os códigos anteriores geralmente geravam 5 linhas nesse nível de otimização, mas como esse código não realizava operações com laço e nem alocação de memória, pode existir alguma relação com esse tamanho de código. Nas duas linhas de código existentes usa MOV para copiar o valor para 0 no registrador e BX juntamente com o LR para fazer o retorno.

  No nível -O2 assim como os exemplos anteriores existem apenas duas linhas de código assembly, uma instrução MOV que registra no R0 o imediato "0" e depois uma chamada de retorno com o comando BX lr, já que o endereço de retorno está ligado no registrador lr.

  No nível -O3, também há apenas duas linhas de comando assembly e o comando é igual ao comando existente no nível de otimização -O2.

- Intel – O código gerado para Intel no exemplo 11 continha um total de 28 linhas quando foi gerado o código assembly com o nível de otimização -O0 (Sem otimização), foi gerado o código completo para o código C inclusive instruções que não serão utilizadas pelo programa.

  No nível de otimização -O1 o compilador gerou apenas 2 linhas, onde há apenas uma instrução MOV colocando o valor imediato 0 no registrador 'eax', e por último uma instrução de retorno.

  No nível de otimização -O2 há uma mudança, o compilador gerou apenas 2 linhas também assim como no nível anterior, porém trocou a instrução MOV po ruma XOR, que compara o registrador 'eax' com ele mesmo, e por último uma instrução de retorno.

  No nível de otimização -O3 o compilador gerou apenas 2 linhas assim como no nível anterior, e as instruções utilizadas foram as mesmas que no nível anterior.

4. Resultados

Após concluir a análise percebe-se que a "lógica" como o compilador gera os códigos assembly para cada arquitetura é parecida, mas há mudanças em nomes de registradores e comandos entre as arquiteturas, é possível perceber também que o compilador utiliza instruções específicas para fazer as operações com double e int. Pelo menos isso é utilizado no nível -O0 de otimização, após isso quando há outras otimizações o compilador reduz drasticamente o código fazendo apenas o que é realmente necessário no código. Há também diferença nas instruções usadas dentro do próprio ARM para

tipos diferentes, algumas instruções são adicionadas com flags dependendo do tipo de variável que é utilizada, além de existirem registradores específicos para trabalhar com esses tipos.

As diferenças notadas dos códigos foram as seguintes:

Do código 7 para o código 8 apareceram algumas instruções específicas das operações com double que foram as instruções 'VLDR.64', 'VSTR.64' e 'VADD.f64'. Além disso foram utilizados os registradores de precisão dupla D6 e D7, o que é normal já que as operações feitas foram com valores de ponto flutuante, e isso pode ser notado ao ver que essas instruções não aparecem no código 7.

Do código 9 para o código 10 apareceram novamente as instruções específicas das operações com double que foram as instruções 'VLDR.64', 'VSTR.64' e 'VADD.f64'. Além disso foram utilizados os registradores de precisão dupla D6 e D7, o que é normal já que as operações feitas foram com valores de ponto flutuante no código 10, enquanto no código 9 não existem essas instruções, também é possível notar a instrução de alocação de memória "Malloc" em ambos os códigos juntamente com a instrução de branch BL.

Do código 11 para o código 12 apareceram novamente as instruções específicas das operações com double que foram as instruções 'VLDR.64', 'VCMPE.f64', 'VSTR.64' e 'VADD.f64'. Além disso foram utilizados os registradores de precisão dupla D6 e D7, o que é normal já que as operações feitas foram com valores de ponto flutuante no código 12, enquanto no código 11 não existem essas instruções, entre todos os códigos esses foram os código que mais apresentaram instruções de desvios, além de serem os códigos que utilizaram instruções específicas de comparação.

Os códigos Intel, assim como os códigos ARM tiveram basicamente as mesmas diferenças, em todas as comparações de código apenas diferenças nos registradores utilizados, algumas instruções que acionavam flags e as diferenças dos registradores específicos de cada arquitetura para ponto flutuante.

5. Referências

GCC -Documentação sobre opções de otimização – Disponível em: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Patterson, David A., and John L. Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.

Pipeline – Disponível em:

https://www.ques10.com/p/34835/explain-pipelining-in-arm-processor/