Lenguaje Ensamblador Risc_V

CAPITULO 1
¿Por qué RISC-V?

1.1 Introducción

El objetivo de RISC-V ("RISC cinco") es convertirse en un ISA universal:

- Debe acoplarse a todos los procesadores, desde los microcontroladores más pequeños para sistemas embebidos, hasta las supercomputadoras más potentes.
- Debe funcionar bien con una amplia variedad de paquetes de software y lenguajes de programación populares.
- Debe poder implementarse en todo tipo de tecnologías.
- Debe ser eficiente para todo tipo de micro-arquitecturas.
- Debe ser estable, implicando que el ISA base no debe cambiar. Aun más importante, no debe descontinuarse.

RISC-V es inusual no solo porque es un ISA reciente—nacido en esta década, mientras que la mayoría de las alternativas nacieron en los 1970s o 1980s— ***pero también porque es un ISA abierto.***

1.2 ISAs Modulares vs. Incrementales

El enfoque convencional en arquitectura de computadoras es desarrollar ISAs incrementales, en los cuales, los nuevos procesadores no solo implementan las nuevas extensiones, sino además todas las instrucciones de ISAs anteriores. El objetivo es mantener compatibilidad binaria para que los programas ya compilados y en formato binario de décadas anteriores, aún puedan funcionar en los procesadores más recientes. Dicho requerimiento, combinado con la ventaja de mercadeo que daba anunciar instrucciones nuevas en cada nueva generación de procesadores, provocó un incremento sustancial en la cantidad y complejidad del ISA. (Es decir, a medida que van saliendo nuevos procesadores y que se van agregando nuevas instrucciones y funcionalidades se van apilando, formando cada vez una ISA más extensa y compleja).

RISC-V es inusual dado que, a diferencia de casi todos los ISAs anteriores, es modular. El núcleo fundamental del ISA es llamado RV32I, el cual ejecuta un stack de software completo. RV32I está congelado y nunca cambiará, lo cual provee un objetivo estable para desarrolladores de compiladores, sistemas operativos y programadores de lenguaje ensamblador. La modularidad viene de extensiones opcionales estándar que el hardware puede incorporar de acuerdo a las necesidades de cada aplicación.

Esta modularidad permite implementaciones muy pequeñas y de bajo consumo energético de RISC-V, lo cual puede ser crítico para aplicaciones embebidas.

1.3 Introducción al Diseño del ISA

Antes de presentar el ISA del RISC-V, será útil entender los principios básicos y los sacrificios/compromisos que debe tomar un arquitecto de computadoras al momento de diseñar un ISA. A continuación, se muestra un listado de las siete métricas:

- costo (ícono de la moneda de un USD)
- simplicidad (rueda)
- rendimiento (odómetro)
- aislamiento de arquitectura e implementación (mitades aisladas de un círculo)
- Tamaño del Programa
- facilidad de programar / compilar / linkear

CAPITULO 2
RV32I: ISA RISC-V Base para Números Enteros

2.2 Formato de Instrucciones RV32I

Son seis formatos de instrucciones básicos:

- tipo-R para operaciones entre registros;
- tipo-I para inmediatos cortos y loads;
- tipo-S para stores; tipo-B para branches;
- tipo-U para inmediatos largos;
- tipo-J para saltos incondicionales.

Incluso el formato de instrucciones muestra ejemplos donde el diseño simple de RISC-V mejora el costo-rendimiento. Primero, únicamente hay seis formatos y todas las instrucciones son de 32 bits, simplificando la decodificación de instrucciones.

Segundo, las instrucciones de RISC-V ofrecen operandos de tres registros, en vez de tener un campo compartido para origen y destino.

Tercero, en RISC-V los bits de los registros a ser leídos y escritos van en la misma posición para todas las instrucciones, implicando que se puede comenzar a acceder a dichos registros antes de la decodificación.

Cuarto, los campos inmediatos en estos formatos siempre son extendidos en signo, y el bit del signo siempre está en el bit más significativo de la instrucción. Esta decisión implica que la extensión de signo del inmediato (lo cual también puede estar en un área crítica en el tiempo), puede continuar antes de la decodificación.

2.3 Registros de RV32I:

Para el agrado de los programadores de ensamblador y compiladores, RV32I tiene 31 registros, más x0, que siempre tiene el valor 0.

Tener un registro a cero tiene un impacto tremendo en simplificar el ISA de RISC-V.

El PC es uno de los 16 registros de ARM-32, lo cual implica que cualquier instrucción que modifica un registro puede ser, como efecto secundario, una instrucción de branch. El PC como registro complica la predicción de branches, lo cual es vital para un buen rendimiento del pipeline, dado que cualquier instrucción puede ser un branch en lugar del 10–20% de las instrucciones típicamente ejecutadas por programas. Además implica un registro de propósito general menos.

2.4 Computación Entera de RV32I:

Las instrucciones aritméticas sencillas (add, sub), instrucciones lógicas (and, or, xor), e instrucciones de corrimiento (sll, srl, sra) son lo que se esperaría de cualquier ISA. Leen dos valores de registros de 32 bits y escriben el resultado al registro destino también de 32 bits. RV32I además ofrece versiones inmediatas de estas instrucciones. A los valores inmediatos siempre se les hace sign-extension, por lo que pueden ser negativos, razón por la cual no hay necesidad de sub.

Los programas pueden generar valores Booleanos del resultado de una comparación. Para permitir dichos casos, RV32I provee la instrucción set less than, la cual guarda un 1 en el registro si el primer operando es menor que el segundo, o 0 en caso contrario. Como es de esperarse, hay una versión con signo (slt) y una sin signo (sltu) para enteros signed y unsigned, así como versiones inmediatas para ambas (slti, sltiu).

RV32I tampoco incluye multiplicación ni división; éstas son parte de la extensión opCosto cional RV32M.

Primero, no hay operaciones enteras para bytes o half-words. Las operaciones siempre son del ancho del registro. Accesos a memoria consumen energía en órdenes de magnitud superiores a operaciones aritméticas, por lo que accesos pequeños a datos pueden ahorrar energía, pero las operaciones aritméticas pequeñas no ahorran.

RV32I también omite instrucciones de rotación y detección de overflow aritmético. Ambas pueden ser calculadas con un par de instrucciones RV32I.

2.5 Loads y Stores de RV32I:

Además de proveer loads y stores de palabras de 32 bits (lw, sw), la Figura 2.1 muestra que RV32I puede cargar bytes y halfwords, ya sea en su versión signed o unsigned (lb, lbu, lh, lhu) y guardar bytes y halfwords (sb, sh). Bytes y halfwords con signo hacen signextension a 32 bits y son escritos en el registro destino. Esta extensión de datos permite que las operaciones aritméticas subsiguientes operen correctamente con 32 bits aun cuando el dato original es más corto. Bytes y halfwords sin signo, útiles para texto y números sin signo, se extienden con cero a 32 bits antes de ser escritos al registro destino.

RV32I omitió los modos de direccionamiento sofisticados de ARM32 y x86-32. Desafortunadamente, todos los modos de direccionamiento de ARM-32 no están disponibles para todos los tipos de datos, pero los modos de direccionamiento de RV32I no discriminan a ningún tipo de dato.

Mientras que ARM-32 y MIPS-32 requieren que los datos estén alineados en memoria, RISC-V no lo exige. Accesos desalineados a veces son requeridos cuando migramos código antiguo. Una opción es no permitir accesos desalineados en el ISA base y proveer instrucciones separadas para soportar accesos desalineados, similar a Load Word Left y Load Word Right de MIPS-32. Sin embargo, esta opción complicaría el acceso a registros, ya que lwl y lwr requieren escribir únicamente a partes de registros en lugar de registros completos. Permitir que los loads y stores pudieran acceder a memoria desalineada simplificaba el diseño general.

2.6 Branches Condicionales de RV32I:

RV32I puede comparar dos registros y saltar si el resultado es igual (beq), distinto (bne), mayor o igual (bge), o menor (blt). Los últimos dos casos son comparaciones con signo, pero RV32I también ofrece versiones sin signo: bgeu y bltu. Las dos relaciones restantes ("mayor que" y "menor o igual") se obtienen intercambiando los argumentos, dado que $x < y$ implica $y > x$ y $x \geq y$ equivale a $y \leq x$.

Dado que las instrucciones de RISC-V deben ser múltiplos de dos bytes el modo de direccionamiento de branches multiplica el valor inmediato de 12 bits por 2, le extiende el signo y lo suma al PC.

2.7 Salto Incondicional de RV32I

La instrucción jump and link (jal) en la Figura 2.1 cumple dos propósitos. En llamadas Simplicidad a funciones, almacena la dirección de la siguiente instrucción PC+4 en el registro destino, normalmente el registro de la dirección de retorno ra (ver Figura 2.4). Para saltos incondicionales, utilizamos el registro cero (x0) en lugar de ra como el registro destino, dado que éste no cambia. Al igual que los branches, jal multiplica su dirección de 20 bits por 2, extiende el signo y suma el resultado al PC para obtener la dirección a saltar.

La versión de jump and link (jalr) con registro es también multipropósito. Puede hacer una llamada a función a una dirección de memoria calculada dinámicamente o simplemente retornar de la función usando a ra como registro origen, y el registro cero (x0) como destino. Enunciados de switch o case, que calculan la dirección a saltar, también pueden usar jalr con el registro cero como destino.

2.10 Observaciones Finales

- Espacio de memoria de 32 bits direccionable por bytes
- Todas las instrucciones son de 32 bits
- 31 registros, todos de 32 bits, y el registro 0 alambrado a cero
- Todas las operaciones son entre registros (ninguna es de registro a memoria)
- Load/Store word, más load/store byte y halfword (signed y unsigned
- Opción de inmediatos en todas las instrucciones aritméticas, lógicas y de corrimientos.
- A los valores inmediatos siempre se les hace sign-extension
- Un modo de direccionamiento (registro + inmediato) y branching relativo al PC
- No hay instrucciones de multiplicación ni división
- Una instrucción que carga un valor inmediato de 20 bits a la parte alta del registro para cargar una constante de 32 bits en 2 instrucciones

CAPITULO 3

Lenguaje Ensamblador RISC-V

3.2 Convención de llamadas

Hay seis etapas generales al llamar una función [Patterson and Hennessy 2017]:

1. Poner los argumentos en algún lugar donde la función pueda acceder a ellos.
2. Saltar a la función (utilizando jal de RV32I).
3. Reservar el espacio de memoria requerido por la función, almacenando los registros que se requiera.
4. Realizar la tarea requerida de la función.
5. Poner el resultado de la función en un lugar accesible por el programa que invocó a la función, restaurando los registros y liberando la memoria.
6. Dado que una función puede ser llamada desde varias partes de un programa, retornar el control al punto de origen (usando ret).

Para obtener un buen rendimiento, es preferible mantener las variables en registros y no en memoria, y, por otro lado, evitar accesos a memoria para guardar y restaurar estos registros.

Afortunadamente, RISC-V tiene suficientes registros para dar lo mejor de ambos mundos: mantener los operandos en registros y reducir la necesidad de guardarlos y restaurarlos. La Rendimiento clave es tener algunos registros que no se garantiza que conserven su valor a través de una llamada a una función, llamados temporary registers, y otros que sí se conservan, llamados saved registers. Funciones que no llaman a otras funciones son llamadas funciones hoja . Cuando una función hoja tiene pocos argumentos y variables locales, podemos guardar todo en registros sin “derramarlos” a memoria. Si estas condiciones se cumplen, el programa no necesita guardar los valores de los registros en memoria, y una fracción sorprendente de llamadas a funciones caen en este afortunado caso.

La Figura 3.2 muestra los nombres de los registros ABI de RISC-V y la convención de preservar a través de llamadas a funciones o no.

Registro	Nombre ABI	Descripción	¿Preservado en llamadas?
x0	zero	Alambrado a cero	—
x1	ra	Dirección de retorno	No
x2	sp	Stack pointer	Sí
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Link register temporal/alterno	No
x6–7	t1–2	Temporales	No
x8	s0/fp	Saved register/frame pointer	Sí
x9	s1	Saved register	Sí
x10–11	a0–1	Argumentos de función/valores de retorno	No
x12–17	a2–7	Argumentos de función	No
x18–27	s2–11	Saved registers	Sí
x28–31	t3–6	Temporales	No
f0–7	ft0–7	Temporales, FP	No
f8–9	fs0–1	Saved registers, FP	Sí
f10–11	fa0–1	Argumentos/valores de retorno, FP	No
f12–17	fa2–7	Argumentos, FP	No
f18–27	fs2–11	Saved registers, FP	Sí
f28–31	ft8–11	Temporales, FP	No

3.3 Ensamblador

La tarea del ensamblador en la Figura 3.1 no es simplemente producir código objeto a partir de instrucciones que el procesador pueda ejecutar, sino además extenderlas para incluir operaciones útiles para el programador de lenguaje ensamblador o el escritor de compiladores. Esta categoría, basada en configuraciones ingeniosas de instrucciones normales es llamada pseudoinstrucciones.

Las Figuras 3.3 y 3.4 enumeran las pseudoinstrucciones de RISC-V. En la primera figura, todas dependen en que el registro x0 siempre sea cero, mientras que en el segundo listado no dependen de eso. Por ejemplo, la instrucción ret mencionada anteriormente es en realidad una pseudoinstrucción que el ensamblador reemplaza por jalr x0, x1, 0 (ver Figura 3.3). La mayoría de las pseudoinstrucciones de RISC-V dependen de x0. Como pueden ver, apartar uno de los 32 registros para que esté alambrado a cero, simplifica significativamente el set de instrucciones de RISC-V permitiendo muchas operaciones populares—tales como: jump, return y branch on equal to zero—como pseudoinstrucciones.

//me ahorre las imágenes pues son los listados de las instrucciones.

Los comandos que comienzan con un punto son directivas del ensamblador. Estos son comandos para el ensamblador y no código a ser traducido. Le indican al ensamblador dónde poner código y datos, especifican constantes de texto y datos para uso en el programa, etcétera.

3.4 Linker

En lugar de compilar todo el código fuente cada vez que cambia un archivo, el linker permite que archivos individuales puedan ser ensamblados por separado. Luego “une” el código objeto nuevo con otros módulos precompilados, tales como librerías. Deriva su nombre a partir de una de sus tareas, la de editar todos los links de las instrucciones de jump and link en el archivo objeto. Además de las instrucciones, cada archivo objeto contiene una tabla de símbolos, la cual contiene todas las etiquetas en el programa que deben ser modificadas como parte del proceso de linking. Esta lista incluye etiquetas al área de datos y código.

3.5 Linking Estático vs Dinámico

La sección anterior describe linking estático, en el cual todo el código potencial de una librería se unifica con el programa y se carga a memoria antes de ejecutarse. Dichas librerías tienden a ser bastante grandes, por lo que unir librerías populares a múltiples programas desperdicia memoria. Además, las librerías se adjuntan al momento del linking—aun cuando luego son actualizadas para arreglar bugs—forzando al programa a utilizar versiones antiguas que pueden tener bugs. Para evitar ambos problemas, la mayoría de los sistemas usan linking dinámico, donde la función externa es cargada y unida al programa luego que se llama por primera vez; si nunca se llama, no es cargada ni linkeada.

3.6 Loader

Un programa como el que aparece en la Figura 3.8 es un archivo ejecutable almacenado. Cuando se desea ejecutar, el trabajo del loader es cargarlo a memoria y saltar a la dirección de inicio. Actualmente, el “loader” es el sistema operativo; dicho de otra manera, cargar a a.out es una de las muchas tareas del sistema operativo.