

KIẾN TRÚC MÁY TÍNH

ET4040

TS. Nguyễn Đức Minh

[Adapted from Computer Organization and Design, 4th Edition, Patterson & Hennessy, © 2008, MK]
[Adapted from Computer Architecture lecture slides, Mary Jane Irwin, © 2008, PennState University]

Điểm số

Điều kiện thi	Lab, Bài tập chương
Bài thi giữa kỳ	30%
Bài tập lớn	20%
Tiến trình	10%
Bài thi cuối kỳ	70%

Tích lũy, trừ qua trả lời câu hỏi trên lớp và đóng góp tổ chức lớp

Tổ chức lớp

Số tín chỉ	3 (3-1-0-6)
Giảng viên	TS. Nguyễn Đức Minh
Văn phòng	611 TV TQB
Email	minh.nguyenduc1@hust.edu.vn
Website	http://ca.edabk.org
Sách	Computer Org and Design, 4th Ed., Patterson & Hennessy, ©2007

Thí nghiệm	3 bài
Bài tập	Theo chương, đề bài xem trên trang web

Giới thiệu

2

HUST-FET, 20/02/2016



Lịch học

- ❖ Thời gian, địa điểm:
 - Lý thuyết: 11 buổi x 135 phút / 1 buổi
 - Bài tập lớn: 4 buổi x 135 phút / 1 buổi
 - Thay đổi lịch (nghỉ, học bù) sẽ được thông báo trên website trước 2 ngày
- ❖ Đăng ký học: ca.edabk.org

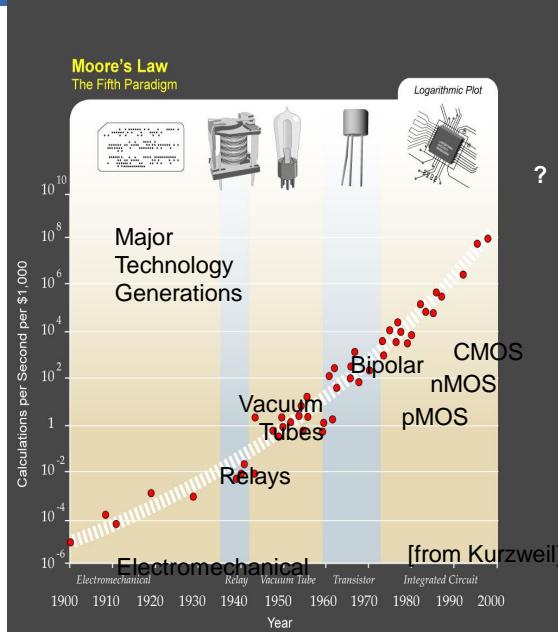


Outline

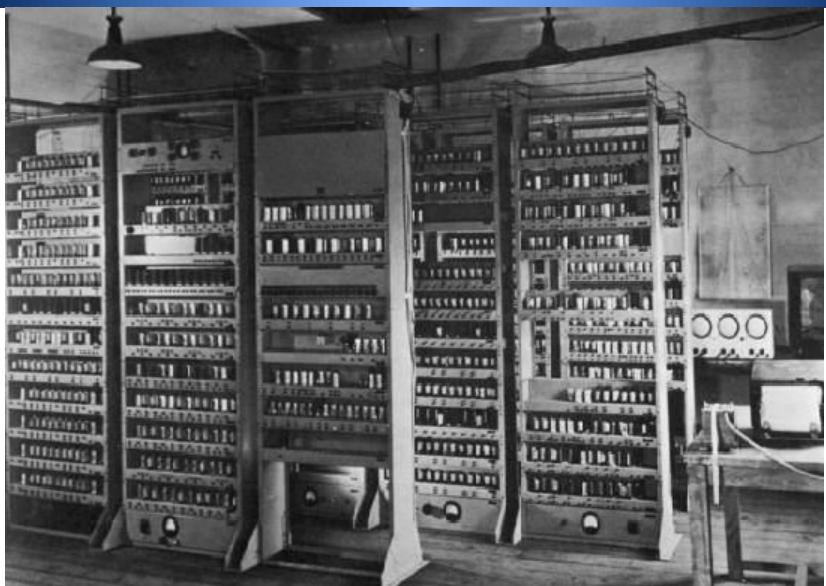
CHƯƠNG 1. GIỚI THIỆU CHUNG VỀ HỆ THỐNG MÁY TÍNH (3 LT+1 BT) (4LT)

1. Xu hướng công nghệ
2. Ứng dụng
3. Phân loại
4. Cấu trúc một hệ thống máy tính-Mô hình von Neumann, mô hình Harvard
5. Khái niệm về hiệu năng hệ thống máy tính (Định nghĩa, Công thức tính, Các yếu tố ảnh hưởng; Chương trình kiểm chuẩn hiệu năng)

Phát triển của công nghệ thông tin



Bắt đầu

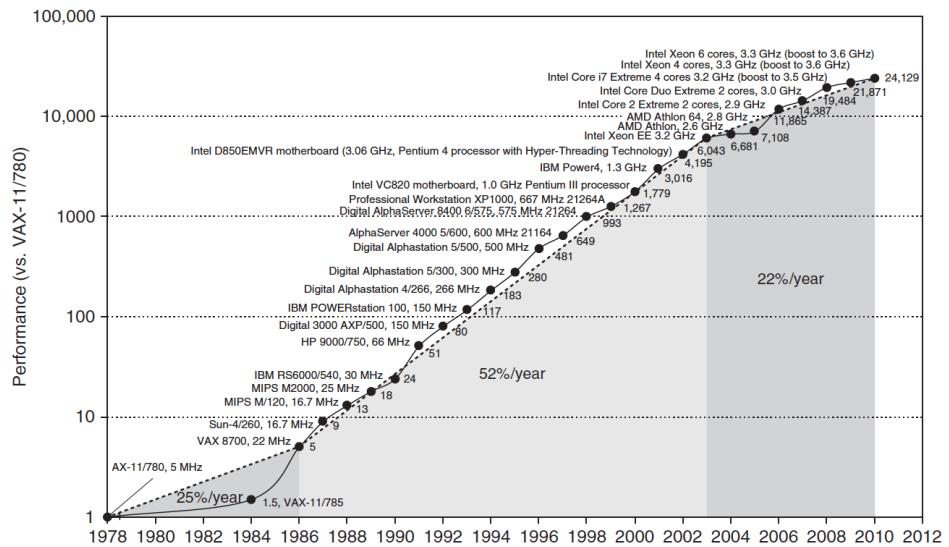


EDSAC, University of Cambridge, UK, 1949

Ngày nay



Máy tính có mặt khắp mọi nơi



Ứng dụng

❖ Phương tiện giao thông

- ❑ Khi máy tính trở nên rẻ hơn, nhỏ hơn và có hiệu suất cao hơn, nó được sử dụng trong ô tô, xe máy để tăng hiệu suất sử dụng nhiên liệu, giảm ô nhiễm, tăng độ an toàn.

❖ Điện thoại di động, thiết bị viễn thông

- ❑ Giúp con người giao tiếp dù ở bất kỳ đâu.

❖ Bản đồ gen

- ❑ Máy tính trở nên rẻ và mạnh hơn 10-100 lần so với cách đây 10 năm cho phép trang bị các máy tính để phân tích và ánh xạ bản đồ gen người.

❖ WWW

- ❑ Nhờ sự phổ biến của máy tính, các thiết bị mạng, Internet trở thành môi trường làm thế giới tràn ngập thông tin (thế giới thông tin).

❖ Bộ tìm kiếm

- ❑ Google trở thành 1 động từ và 1 đế chế.

Các loại máy tính

❖ Máy tính để bàn (eng, Desktop computers)

- ❑ Một người dùng; Chạy nhiều ứng dụng khác nhau; Đi kèm màn hình, bàn phím và chuột; Yêu cầu giá thành rẻ, hiệu năng cao

❖ Máy chủ (eng, Servers)

- ❑ Nhiều người dùng đồng thời; Chạy các ứng dụng lớn; Truy cập qua mạng, Yêu cầu độ ổn định và an toàn cao.

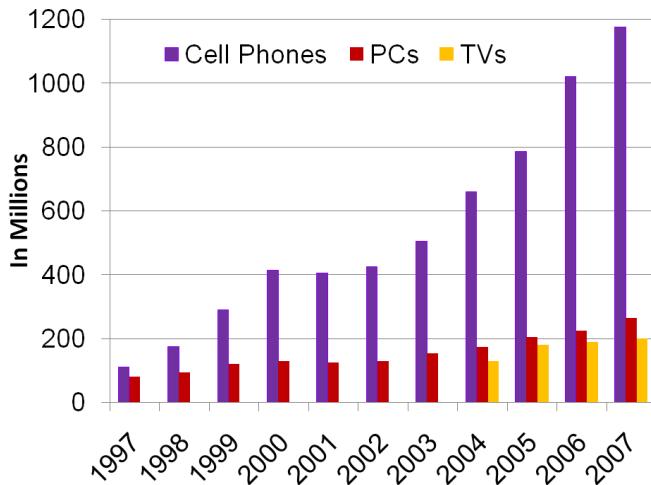
❖ Siêu máy tính (eng, Supercomputers)

- ❑ Chạy các ứng dụng khoa học và công nghệ cao cấp; Gồm hàng trăm/nghìn bộ xử lý, bộ nhớ và bộ lưu trữ dung lượng lớn; Yêu cầu hiệu năng cao và có giá thành cao.

❖ Máy tính nhúng (eng, Embedded computers (processors))

- ❑ Máy tính nằm bên trong một thiết bị khác, chạy 1 ứng dụng xác định trước.

Tăng trưởng doanh số điện thoại di động



Tăng trưởng điện thoại di động >> Tăng trưởng máy tính để bàn

- ❖ Ứng dụng trong nhiều lĩnh vực khác nhau
- ❖ Yêu cầu hiệu năng rất khác nhau
- ❖ Yêu cầu hiệu năng tối thiểu và vừa đủ. Ví dụ?
- ❖ Yêu cầu khắt khe về giá thành. Ví dụ?
- ❖ Yêu cầu khắt khe về năng lượng tiêu thụ. Ví dụ?
- ❖ Ít chấp nhận hỏng hóc. Ví dụ?

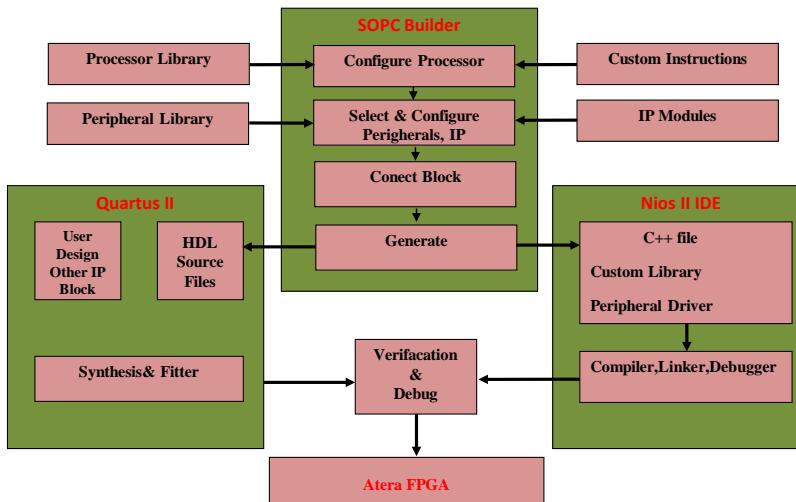


Mục tiêu môn học

Mục tiêu môn học

Kiến thức về hệ thống máy tính:

- ❑ Giao diện giữa phần mềm và phần cứng
- ❑ Quá trình biên dịch chương trình phần mềm
- ❑ Cấu tạo và hoạt động của phần cứng máy tính
- ❑ Phương pháp đánh giá định lượng về hiệu năng máy tính
- ❑ Ảnh hưởng của các thành phần lên hiệu năng máy tính
- ❖ Kỹ sư phần mềm: tận dụng ưu điểm của phần cứng và lựa chọn phần cứng tối ưu
- ❖ Kỹ sư phần cứng: ảnh hưởng của phần cứng lên phần mềm



KIẾN TRÚC MÁY TÍNH

Thành phần cơ bản của máy tính

[Adapted from Computer Organization and Design, 4th Edition, Patterson & Hennessy, © 2008, MK]
 [Adapted from Computer Architecture lecture slides, Mary Jane Irwin, © 2008, PennState University]



Outline

CHƯƠNG 1. GIỚI THIỆU CHUNG VỀ HỆ THỐNG MÁY TÍNH (3 LT+1 BT) (4LT)

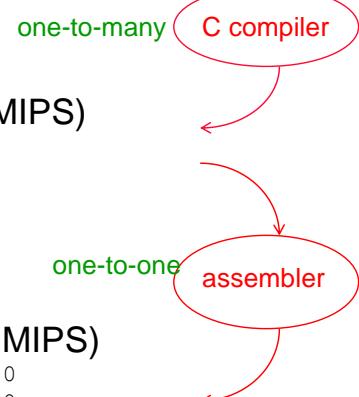
1. Xu hướng công nghệ
2. Ứng dụng
3. Phân loại
4. Cấu trúc một hệ thống máy tính-Mô hình von Neumann, mô hình Havard
5. Khái niệm về hiệu năng hệ thống máy tính (Định nghĩa, Công thức tính, Các yếu tố ảnh hưởng; Chương trình kiểm chuẩn hiệu năng)



Trình biên dịch

❖ High-level language program (in C)

```
swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



❖ Assembly language program (for MIPS)

```
swap: sll $2, $5, 2
      add $2, $4, $2
      lw $15, 0($2)
      lw $16, 4($2)
      sw $16, 0($2)
      sw $15, 4($2)
      jr $31
```

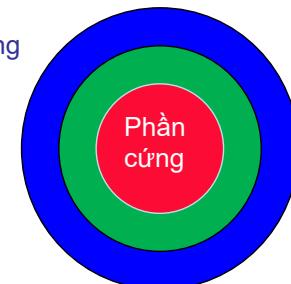
❖ Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
, , ,
```



Phần mềm

Phần mềm ứng dụng



Phần mềm hệ thống

❖ Phần mềm hệ thống

- ❑ Hệ điều hành – giám sát, giao tiếp giữa phần cứng và phần mềm ứng dụng (như Linux, MacOS, Windows)
 - Điều khiển các hoạt động vào ra cơ bản
 - Cấp phát bộ nhớ
 - Cung cấp sự chia sẻ có bảo vệ giữa các ứng dụng
- ❑ Bộ biên dịch – chuyển đổi các chương trình ở ngôn ngữ bậc cao (như C, Java) thành các câu lệnh phần cứng có thể thực hiện



Ưu điểm của ngôn ngữ bậc cao

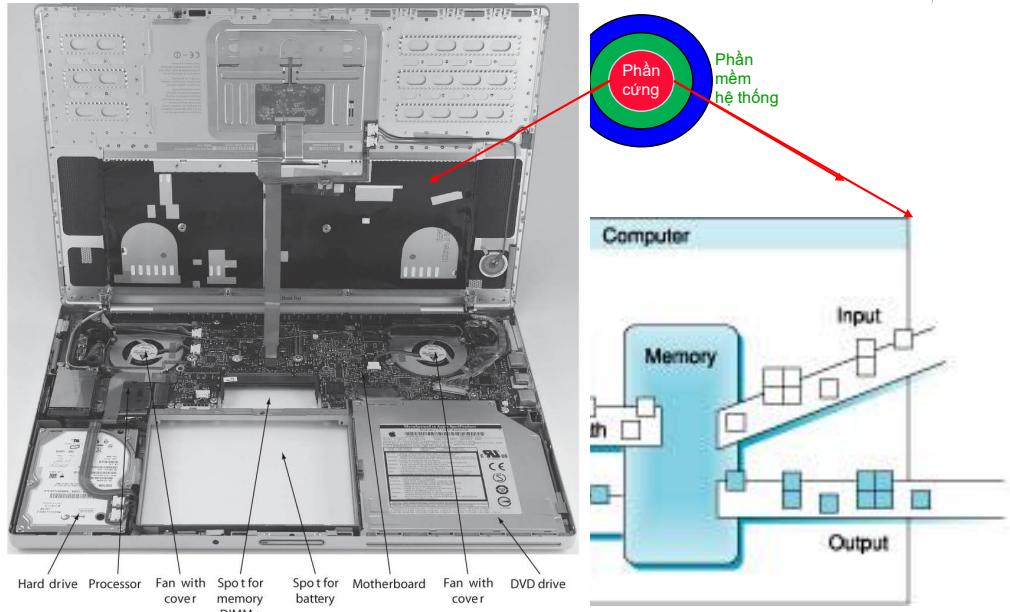
❖ Ngôn ngữ bậc cao

- ❑ Chương trình được viết ở ngôn ngữ tự nhiên và phù hợp với từng ứng dụng (Ví dụ: Fotran, Lisp, Java)
- ❑ Tăng năng suất lập trình viên – mã chương trình dễ hiểu, dễ gỡ lỗi, dễ kiểm tra
- ❑ Tăng khả năng bảo trì chương trình
- ❑ Chương trình độc lập với phần cứng sẽ thực hiện chương trình
- ❑ Chương trình được tối ưu hóa cho từng loại phần cứng nhờ các thuật toán tối ưu trong trình biên dịch

❖ Ít chương trình còn được phát triển bằng hợp ngữ



Phản cứng

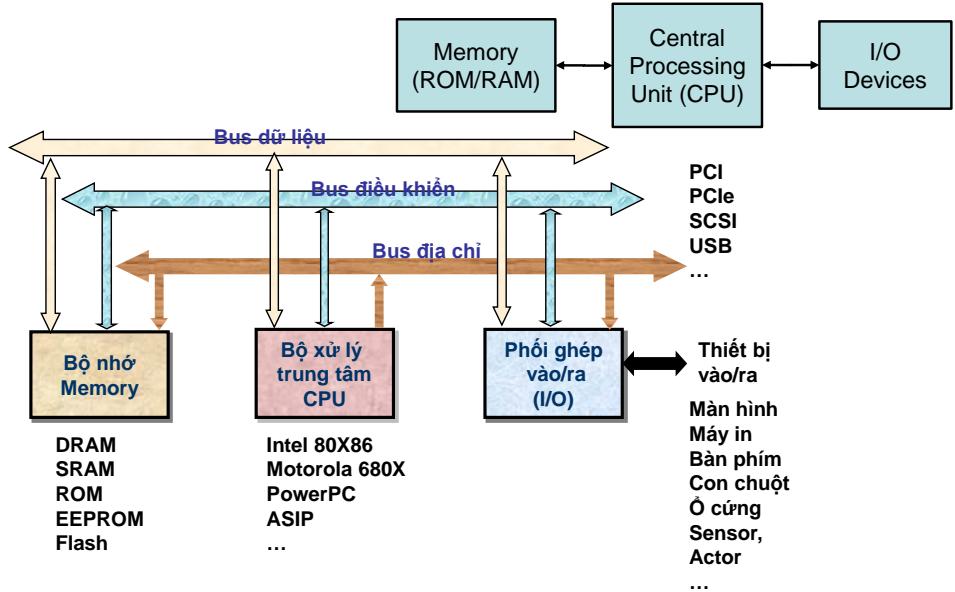


Chương 1 – Thành phần cơ bản của máy tính

21

HUST-FET, 20/02/2016

Kiến trúc vonNeumann

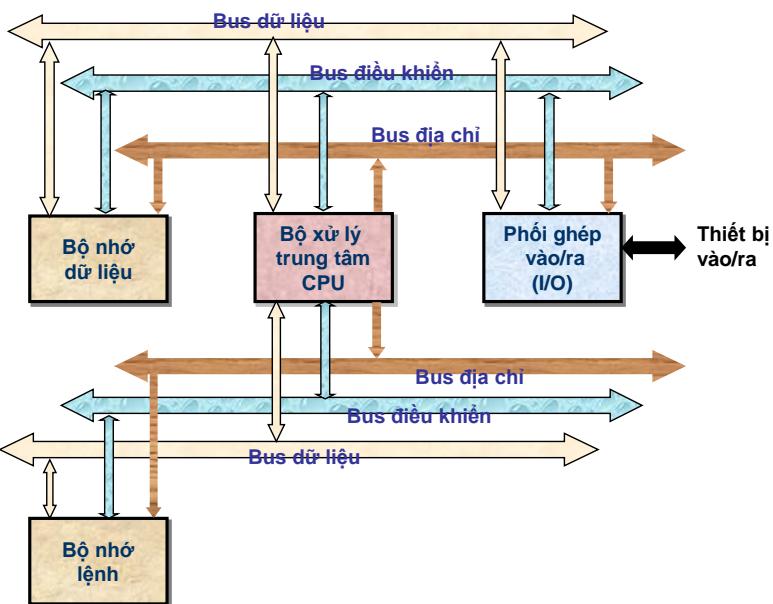


Chương 1 – Thành phần cơ bản của máy tính

22

HUST-FET, 20/02/2016

Kiến trúc Harvard

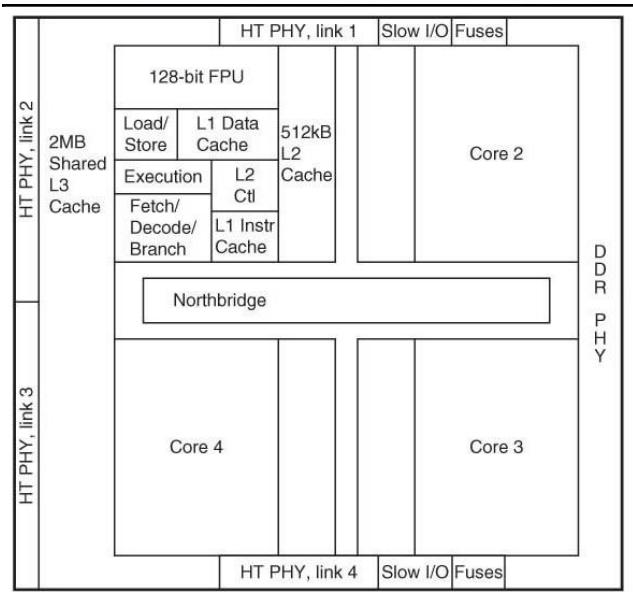


Chương 1 – Thành phần cơ bản của máy tính

23

HUST-FET, 20/02/2016

Bộ xử lý đa nhân AMD's Barcelona



Chương 1 – Thành phần cơ bản của máy tính

24

HUST-FET, 20/02/2016

- ❑ 4 nhân, lệnh không theo thứ tự
- ❑ Đồng hồ: 1,9 GHz
- ❑ Công nghệ 65nm
- ❑ 3 mức bộ đệm (L1, L2, L3)
- ❑ Tích hợp bộ điều khiển cầu bắc

- ❖ Kiến trúc tập lệnh (*eng, Instruction Set Architecture - ISA*), hay kiến trúc: là giao diện trừu tượng giữa phần cứng và các phần mềm ở lớp thấp nhất, bao gồm tất cả các thông tin cần thiết để viết chương trình ở ngôn ngữ máy như: các chỉ thị máy, thanh ghi, bản đồ bộ nhớ, phương pháp vào ra, ...
- ❖ Giao diện nhị phân ứng dụng (*eng, Application Binary Interface – ABI*) bao gồm các chỉ thị máy cơ bản người dùng có thể sử dụng và các hàm hệ thống cấp thấp do hệ điều hành cung cấp.

Kết luận:

- Máy tính gồm các lớp phân cấp theo mức độ trừu tượng.
- Kiến trúc tập lệnh là một lớp then chốt trong hệ thống máy tính.
- Các triển khai phần cứng khác nhau tuân theo cùng chuẩn về kiến trúc tập lệnh có thể thực hiện cùng một phần mềm giống nhau.



Hiệu năng (*eng. Performance*)

- ❖ Thời gian đáp ứng (thời gian thực thi) – là khoảng thời gian giữa thời điểm bắt đầu thực hiện và thời điểm hoàn thành một nhiệm vụ
 - Quan trọng đối với 1 người sử dụng yêu cầu hệ thống thực hiện 1 nhiệm vụ
- ❖ Thông lượng (dải thông) – là tổng số nhiệm vụ có thể được hoàn thành trong 1 khoảng thời gian
 - Quan trọng đối với người điều hành trung tâm dữ liệu
- ❖ Cần các hệ đo lường khác nhau cho hiệu năng của máy tính cũng như cần 1 tập hợp các ứng dụng khác nhau để kiểm chuẩn các máy tính nhúng, máy tính để bàn (thường chú trọng đến thời gian đáp ứng) và các máy chủ (thường chú trọng đến thông lượng)



- ❖ Quyết định mua máy tính
 - Trong số các máy tính, máy nào có
 - > hiệu năng tốt nhất?
 - > giá thành rẻ nhất?
 - > tỉ lệ giá thành/hiệu năng tốt nhất?
- ❖ Lựa chọn thiết kế máy tính
 - Trong các lựa chọn thiết kế, thiết kế nào
 - > cho cải tiến tốt nhất về hiệu năng?
 - > giá thành thấp nhất?
 - > tỉ lệ giá thành/hiệu năng tốt nhất?
- ❖ Yêu cầu:
 - Căn cứ để so sánh
 - Thông số đánh giá
- ❖ Mục tiêu: nắm rõ sự
 - ảnh hưởng của các nhân tố trong kiến trúc máy tính tới hiệu năng toàn hệ thống;
 - vai trò quan trọng tương đối và giá thành của các nhân tố đó,



Ví dụ 1.1 – Cải tiến hiệu năng

- ❖ Ảnh hưởng của bộ xử lý lên thời gian đáp ứng và thông lượng
- ❖ Nếu ta thay đổi cấu trúc máy tính như sau thì thời gian đáp ứng và thông lượng của máy tính thay đổi thế nào?
 - Thay thế bộ xử lý bằng bộ xử lý nhanh hơn
 - Bổ xung 1 bộ xử lý để thực hiện các nhiệm vụ tách biệt (như trong hệ thống tìm kiếm WWW)



- ❖ Hiệu năng (tốc độ) của máy tính X:

$$\text{Performance}_x = \frac{1}{\text{Execution Time}_x}$$

- ❖ Máy tính X nhanh hơn máy tính Y, n lần:

$$\frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{Execution Time}_y}{\text{Execution Time}_x} = n$$

- ❖ Để tối đa hóa hiệu năng, cần tối thiểu hóa thời gian thực hiện
- ❖ Giảm thời gian đáp ứng thường sẽ tăng thông lượng



Ví dụ 1.2 – So sánh hiệu năng

- ❖ Nếu máy tính A thực hiện 1 chương trình mất 10s và máy tính B chạy cùng chương trình đó mất 15s, máy tính A nhanh hơn máy tính B bao nhiêu lần?
- ❖ Tỉ lệ so sánh hiệu năng của A và B là:

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A} = \frac{15}{10} = 1,5$$

➔ A nhanh hơn B 1,5 lần



Ví dụ 1.2 – So sánh hiệu năng

- ❖ Nếu máy tính A thực hiện 1 chương trình mất 10s và máy tính B chạy cùng chương trình đó mất 15s, máy tính A nhanh hơn máy tính B bao nhiêu lần?



Đo hiệu năng – Đo thời gian thực hiện

- Có 3 loại thời gian dùng để tính hiệu năng
- ❖ Thời gian đáp ứng (thời gian đồng hồ, thời gian đã trôi qua):
 - Tổng thời gian hoàn thành 1 nhiệm vụ
 - Bao gồm: thời gian truy cập đĩa, bộ nhớ, thời gian vào ra, thời gian cho hệ điều hành
 - ❖ Thời gian bộ xử lý (CPU time)
 - Thời gian CPU người dùng
 - Thời gian CPU hệ thống



Các yếu tố tính hiệu năng

- ❖ Thời gian CPU – thời gian bộ xử lý dùng để thực hiện 1 nhiệm vụ

- Không bao gồm thời gian chờ vào/ra hay thời gian thực hiện các chương trình khác
- Thời gian CPU cho 1 chương trình, T_{cpu} được tính từ số chu kỳ đồng hồ CPU thực hiện chương trình P và thời gian 1 chu kỳ đồng hồ:

$$T_{cpu} = C \times T_c \text{ or } T_{cpu} = C / f_c$$

- ❖ Hiệu năng có thể cải thiện bằng cách giảm số chu kỳ 1 xung đồng hồ hoặc giảm số chu kỳ cần thiết để thực hiện chương trình



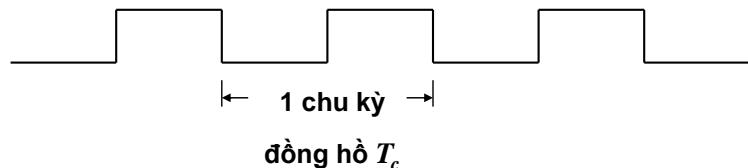
Ví dụ 1.3 – Cải thiện hiệu năng

- ❖ Máy tính A với xung đồng hồ 2GHz thực hiện 1 chương trình hết 10 giây. Để thực hiện chương trình đó trong 6 giây bằng máy tính B, ta cần tăng tốc độ xung đồng hồ của máy B. Tuy nhiên, tăng tốc độ xung đồng hồ cũng làm tăng số chu kỳ cần thiết lên 1,2 lần. Xác định tốc độ xung đồng hồ máy tính B.



Xung nhịp đồng hồ

- ❖ CPU hoạt động đồng bộ theo đồng hồ



10 nsec clock cycle => 100 MHz clock rate

5 nsec clock cycle => 200 MHz clock rate

2 nsec clock cycle => 500 MHz clock rate

1 nsec (10^{-9}) clock cycle => 1 GHz (10^9) clock rate

500 psec clock cycle => 2 GHz clock rate



Ví dụ 1.3 – Cải thiện hiệu năng

- ❖ Công thức để tính thời gian CPU, khi máy tính A thực hiện chương trình:

$$T_{cpu,A} = \frac{C_A}{f_{c,A}}$$

- ❖ Số chu kỳ máy tính A dùng để thực hiện chương trình:

$$C_A = T_{cpu,A} \times f_{c,A} = 10 \times 2 \times 10^9 = 20 \times 10^9 \text{ cycles}$$

- ❖ Số chu kỳ máy tính B dùng để thực hiện chương trình:

$$C_B = 1,2 \times C_A$$

- ❖ Tốc độ đồng hồ của máy tính B:

$$f_{c,B} = \frac{C_B}{T_{cpu,B}} = \frac{1,2 \times 20 \times 10^9}{6} = 4 \text{GHz}$$



Số xung đồng hồ

❖ Số xung đồng hồ thực hiện 1 chương trình:

❖ Trong đó: $C = I \times CPI$

- I là số chỉ thị máy cần thực hiện trong chương trình
- CPI (*eng. Clock cycles per Instruction*) là số xung đồng hồ trung bình cần để thực thi 1 chỉ thị máy,

❖ CPI có thể dùng để so sánh các máy tính khác nhau cùng triển khai 1 kiến trúc tập lệnh.

❖ Ví dụ: có 3 loại lệnh A, B, C khác nhau trong 1 kiến trúc tập lệnh. Mỗi lệnh trong từng loại có CPI tương ứng:

	CPI for this instruction class		
	A	B	C
CPI	1	2	3



Ví dụ 1.4 – So sánh dựa trên CPI

❖ Khi 2 máy tính A, B cùng thực hiện 1 chương trình, chúng cùng thực hiện I chỉ thị. Do đó:

$$T_{cpu,A} = I \times CPI_A \times T_{c,A} = I \times 2,0 \times 250ps = 500 \times I$$

$$T_{cpu,B} = I \times CPI_B \times T_{c,B} = I \times 1,2 \times 500ps = 600 \times I$$

❖ Vì vậy, máy A nhanh hơn máy B và:

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{T_{cpu,B}}{T_{cpu,A}} = \frac{600 \times I}{500 \times I} = 1,2$$

Ví dụ 1.4 – So sánh dựa trên CPI

❖ Máy tính A và B cùng triển khai 1 kiến trúc tập lệnh. Máy A có chu kỳ đồng hồ là 250ps, và CPI hiệu dụng cho 1 chương trình P là 2,0. Máy B có chu kỳ đồng hồ là 500ps, và CPI hiệu dụng cho cùng 1 chương trình P là 1,2. Máy tính nào nhanh hơn và nhanh hơn bao nhiêu?



CPI hiệu dụng (trung bình)

❖ CPI hiệu dụng được tính bằng cách xét tất cả các lớp chỉ thị có trong chương trình và lấy trung bình với trọng số là tỉ lệ xuất hiện của lớp chỉ thị trong chương trình

$$CPI = \sum_{i=1}^n (CPI_i \times IC_i)$$

❖ Trong đó:

- IC_i là tỉ lệ (%) số chỉ thị thuộc lớp i được thực thi
- CPI_i là số chu kỳ (trung bình) cần để thực hiện 1 chỉ thị thuộc lớp i
- N là số lớp chỉ thị

❖ CPI hiệu dụng phụ thuộc vào tỉ lệ chỉ thị trong một chương trình (tần suất động của các chỉ thị trong 1 hoặc nhiều chương trình)



Công thức hiệu năng

❖ Công thức hiệu năng cơ bản:

$$T_{cpu} = I \times CPI \times T_c$$

$$T_{cpu} = \frac{I \times CPI}{f_c}$$

❖ Công thức trên phân tách 3 yếu tố ảnh hưởng đến hiệu năng máy tính

❖ Cho phép đo CPU được đo bằng cách chạy chương trình:

- Tốc độ đồng hồ được cho trước
- Số chỉ thị I được đo bằng cách dùng công cụ profilers/mô phỏng sự thực hiện chương trình mà không cần triển khai phần cứng
- CPI phụ thuộc vào loại chỉ thị, triển khai phần cứng chi tiết

❖ Cho phép so sánh 2 triển khai phần cứng hoặc đánh giá lựa chọn giữa 2 thiết kế

Các yếu tố quyết định hiệu năng CPU

$$T_{cpu} = I \times CPI \times T_c$$

	I	CPI	T _c
Algorithm			
Programming language			
Compiler			
ISA			
Core organization			
Technology			



Các yếu tố quyết định hiệu năng CPU

$$T_{cpu} = I \times CPI \times T_c$$

	I	CPI	T _c
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	X
Core organization		X	X
Technology			X

Ví dụ 1.5 – So sánh đoạn mã chương trình

❖ Người thiết kế một máy tính triển khai kiến trúc tập lệnh gồm 3 loại chỉ thị A, B, C được CPI như sau:

	A	B	C
CPI	1	2	3

❖ Với 1 câu lệnh ở ngôn ngữ bậc cao, người viết trình biên dịch có thể lựa chọn 2 đoạn chỉ thị máy gồm có tần suất các loại chỉ thị như sau:

Đoạn mã	A	B	C
1	2	1	2
2	4	1	1

❖ Đoạn mã nào gồm nhiều chỉ thị hơn? Đoạn mã nào nhanh hơn? Tính CPI của từng đoạn mã.

Ví dụ 1.5 – So sánh đoạn mã chương trình

Đoạn mã	A	B	C
1	2	1	2
2	4	1	1

	A	B	C
CPI	1	2	3

- ❖ Đoạn mã 1 dùng 5 chỉ thị, đoạn mã 2 dùng 6 chỉ thị
- ❖ Số xung đồng hồ để thực hiện mỗi đoạn mã được tính như sau:

$$C_1 = \sum_{i=1}^3 (CPI_i \times I_{1,i}) = (1 \times 2 + 2 \times 1 + 3 \times 2) = 10$$

$$C_2 = \sum_{i=1}^3 (CPI_i \times I_{2,i}) = (1 \times 4 + 2 \times 1 + 3 \times 1) = 9$$

□ Trong đó $I_{1,i}$, $I_{2,i}$ là số lượng chỉ thị loại i trong đoạn mã 1 và 2 tương ứng

- ❖ Như vậy đoạn mã 1 chậm hơn đoạn mã 2, mặc dù dùng ít chỉ thị hơn



Ví dụ 1.6 – Cải tiến hiệu năng

Op	Tần suất (IC_i)	CPI _i	$IC_i \times CPI_i$
ALU	50%	1	0,5
Load	20%	5	1,0
Store	10%	3	0,3
Branch	20%	2	0,4
	$\Sigma =$	2,2	1,95

- ❖ Nếu ta có bộ đệm dữ liệu làm giảm thời gian nạp (Load) xuống 2 chu kỳ, máy tính sẽ nhanh lên bao nhiêu lần?

$$T_{cpu} = 1,6 \times I \times T_c \text{ vì vậy } 2,2/1,6 \text{ means } 37,5\% \text{ faster}$$

- ❖ Nếu ta có khối dự báo rẽ nhánh cho phép tiết kiệm 1 chu kỳ khi rẽ nhánh, hiệu năng sẽ thế nào?

$$T_{cpu} = 2,0 \times I \times T_c \text{ vì vậy } 2,2/2,0 \text{ means } 10\% \text{ faster}$$

- ❖ Nếu ta có 2 khối ALU thực hiện 2 chỉ thị ALU đồng thời?

$$T_{cpu} = 1,95 \times I \times T_c \text{ vì vậy } 2,2/1,95 \text{ means } 12,8\% \text{ faster}$$



Ví dụ 1.6 – Cải tiến hiệu năng

- ❖ Cho một máy tính thực hiện 1 chương trình gồm 4 loại chỉ thị máy có các thông số về tần suất và CPI như sau:

Op	Tần suất (IC_i)	CPI _i	$IC_i \times CPI_i$
ALU		50%	1
Load		20%	5
Store		10%	3
Branch		20%	2
			$\Sigma =$

- ❖ Nếu ta có bộ đệm dữ liệu làm giảm thời gian nạp (Load) xuống 2 chu kỳ, máy tính sẽ nhanh lên bao nhiêu lần?

- ❖ Nếu ta có khối dự báo rẽ nhánh cho phép tiết kiệm 1 chu kỳ khi rẽ nhánh, hiệu năng sẽ thế nào?

- ❖ Nếu ta có 2 khối ALU thực hiện 2 chỉ thị ALU đồng thời?



Chương trình đo kiểm chuẩn

- ❖ Benchmarks – là tập hợp các chương trình tạo nên 1 “workload” được chọn để đo hiệu năng

- ❖ SPEC (System Performance Evaluation Cooperative) tạo ra 1 tập các benchmark chuẩn bắt đầu từ SPEC89. Chuẩn mới nhất là SPEC CPU2006 gồm 12 chương trình số nguyên (CINT2006) và 17 chương trình số thực (CFP2006)

www.spec.org

- ❖ Ngoài ra, còn có các tập các workload để đo kiểm năng lượng (SPECpower_ssj2008), mail (SPECmail2008), hay đa phương tiện (mediabench), ...



Các chương trình đo kiểm



Integer benchmarks		FP benchmarks	
gzip	compression	wupwise	Quantum chromodynamics
vpr	FPGA place & route	swim	Shallow water model
gcc	GNU C compiler	mgrid	Multigrid solver in 3D fields
mcf	Combinatorial optimization	applu	Parabolic/elliptic pde
crafty	Chess program	mesa	3D graphics library
parser	Word processing program	galgel	Computational fluid dynamics
eon	Computer visualization	art	Image recognition (NN)
perlbench	perl application	quake	Seismic wave propagation simulation
gap	Group theory interpreter	facerec	Facial image recognition
vortex	Object oriented database	ammp	Computational chemistry
bzip2	compression	lucas	Primality testing
twolf	Circuit place & route	fma3d	Crash simulation fem
		sixtrack	Nuclear physics accel
		apsi	Pollutant distribution

Chương 1 – Thành phần cơ bản của máy tính

So sánh và tổng kết hiệu năng



❖ Tổng kết hiệu năng cho 1 tập tiêu chuẩn thành một số duy nhất:

- ❑ Thời gian thực hiện được chuẩn hóa thành SPECRatio (tỉ số lớn có nghĩa là nhanh hơn)
- ❑ Lấy trung bình nhân của các SPECRatio:

$$GM = \sqrt[n]{\prod_{i=1}^n \text{SPEC ratio}_i}$$

Chương 1 – Thành phần cơ bản của máy tính

SPEC CINT2006 on Barcelona ($T_c = 0,4 \times 10^{-9}$)



Name	Ix10 ⁹	CPI	ExTime	RefTime	SPEC ratio
perl	2 118	0,75	637	9 770	15,3
bzip2	2 389	0,85	817	9 650	11,8
gcc	1 050	1,72	724	8 050	11,1
mcf	336	10,00	1 345	9 120	6,8
go	1 658	1,09	721	10 490	14,6
hmmer	2 783	0,80	890	9 330	10,5
sjeng	2 176	0,96	837	12 100	14,5
libquantum	1 623	1,61	1 047	20 720	19,8
h264avc	3 102	0,80	993	22 130	22,3
omnetpp	587	2,94	690	6 250	9,1
astar	1 082	1,79	773	7 020	9,1
xalancbmk	1 058	2,70	1 143	6 900	6,0
Geometric Mean					11,7

Chương 1 – Thành phần cơ bản của máy tính



Các thông số hiệu năng khác



❖ Năng lượng tiêu thụ là một thông số quan trọng, đặc biệt với thị trường hệ nhúng (thời lượng pin là quan trọng với hệ thống)

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100 %	231,86 7	295
90%	211,282	286
80%	185,80 3	275
70%	163,42 7	265
60%	140,16 0	256
50%	118,32 4	246
40%	92,03 5	233
30%	70,50 0	222
20%	47,12 6	206
10%	23,06 6	180
0%	0	141
Overall Sum	1,283, 590	2,605
$\Sigma ssj_ops / \Sigma power =$		493

Chương 1 – Thành phần cơ bản của máy tính



Luật Amdahl

- ❖ Khi thực hiện cải tiến một đặc điểm của hệ thống, tác dụng của việc cải tiến bị giới hạn bởi đặc điểm được cải tiến

$$T_{cpu} \text{ sau khi cải tiến} = \frac{T_{cpu} \text{ ảnh hưởng bởi sự cải tiến}}{\text{Tỉ lệ cải tiến}} + T_{cpu} \text{ không ảnh hưởng}$$

- ❖ Luật cơ bản để tính toán định lượng sự cải tiến
- ❖ Khi cải tiến, cần chú trọng đến các trường hợp thông dụng
- ❖ Đặt ra giới hạn số lượng bộ xử lý hoạt động song song

Ví dụ 1.7 - Luật Amdahl

- ❖ 1 chương trình thực hiện bởi 1 máy tính trong 100 giây. Trong đó, 80 giây được dùng để thực hiện phép nhân. Vậy cần phải tăng tốc độ phép nhân lên mấy lần để có thể tăng tốc độ thực hiện chương trình lên 5 lần?

$$T_{cpu} \text{ sau khi cải tiến} = \frac{T_{cpu} \text{ ảnh hưởng bởi sự cải tiến}}{\text{Tỉ lệ cải tiến}} + T_{cpu} \text{ không ảnh hưởng}$$

$$20 \text{ giây} = \frac{80 \text{ giây}}{n} + (100-80)$$

- ❖ Không thể cải tiến để tăng tốc được 5 lần



Kết luận chương

- ❖ Hệ thống máy tính được xây dựng từ **phân cấp các lớp trùu tượng**. Các chi tiết triển khai lớp dưới bị che khuất khỏi lớp trên.
- ❖ **Kiến trúc tập lệnh** – lớp giao tiếp giữa phần cứng và phần mềm mức thấp – là lớp trùu tượng quan trọng trong hệ thống máy tính.
- ❖ Phần cứng máy tính gồm **5 thành phần**: **đường dữ liệu, khối điều khiển, bộ nhớ, khôi vào, và khôi ra**. 5 thành phần đó kết nối với nhau bằng hệ thống bus theo mô hình **vonNeumann** hoặc mô hình **Havard**.
- ❖ Phương pháp đánh giá hiệu năng một hệ thống máy tính là dùng **thời gian thực hiện 1 chương trình**. Thời gian thực hiện chương trình được tính bằng công thức:

$$T_{cpu} = I \times CPI \times T_c$$



Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great Ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolic, © 2020, UC Berkeley]



What you need to know about this class

- Course information

- Course Website:

<https://sites.google.com/site/kimhuetadtvtbk/cac-mon-giang-day/kien-truc-may-tinh>

<https://cs61c.org/sp20/#lectures>

- Textbooks: Average 15 pages of reading/week

- Patterson & Hennessy, Computer Organization and Design, RISC-V edition

- Kernighan & Ritchie, The C Programming Language 2nd Edition

- Barroso & Holzle, The Datacenter as a Computer 3rd

8/28/2021

2

What you need to know about this class

- Course Grading

- Homework, Assignments, mini-tests(15%)

- Project, midterm exam (15%)

- Final exam (70%)

- Extra Score: EPA!

- Effort

- Completing all assignments and homework in time

- Participation

- Asking/Answering Qs in TEAMS/ offline class & making it interactive

- Altruism

- Helping others in class

- Writing software, tutorials that help others learn

- EPA! can bump students up to the next grade level

8/28/2021

8/28/2021

3

What is expected to this course ?

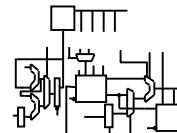
- To explain what's inside the revolutionary machine, unraveling the software below your program and the hardware under the covers of your computer
- To understand the aspects of both the hardware and software that affect program performance
- By the time you complete this course, you will be able to answer the following questions:
 - ✓ How are programs written in a high-level language, such as C, Java translated into the language of hardware
 - ✓ How does the hardware execute the resulting program?
 - ✓ What determines the performance of a program?
 - ✓ What techniques can be used by a programmer/hardware designers to improve the performance

8/28/2021

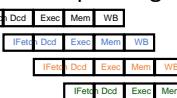
4

Road map

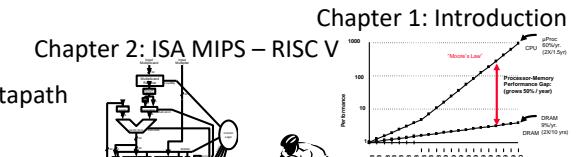
Chapter 3: Single/multicycle Datapath



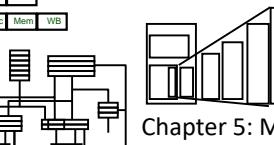
Chapter 4: Pipelining



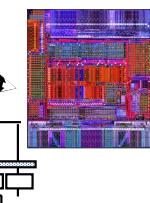
8/28/2021



Course information is described in more detail in the syllabus



Chapter 5: Memory Systems



Chapter 6: Parallel processor

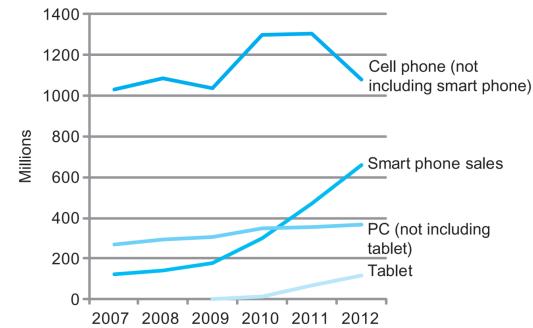
Introduction to Computer Abstractions and Technology

- Introduction
- Great Ideas in Computer Architecture

8/28/2021

6

Introduction



- Hardware advances have allowed programmers to create wonderfully useful software, which explains why computers are omnipresent
- Computers in automobiles
 - Cell phones
 - Human genome project
 - World Wide Web
 - Search engines

Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011. Tablets are the fastest growing category, nearly doubling between 2011 and 2012. Recent PCs and traditional cell phone categories are relatively flat or declining

8/28/2021

7

Welcome to the Post-PC Era



Embedded computes in Network Edge Devices



My other computer is a data center

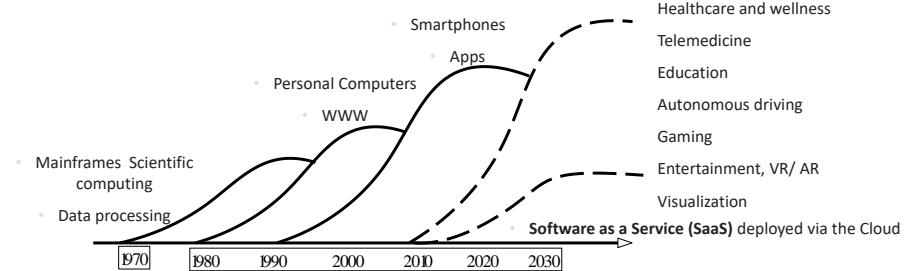
Replacing the PC is the Personal Mobile Devices (PMD)

Taking over from the conventional server is **Cloud Computing**, which relies upon giant datacenters that are now known as *Warehouse Scale Computers* (WSCs). Companies like Amazon and Google build these WSCs containing 100,000 servers

8/28/2021

8

Why is Computer Architecture Exciting Today?

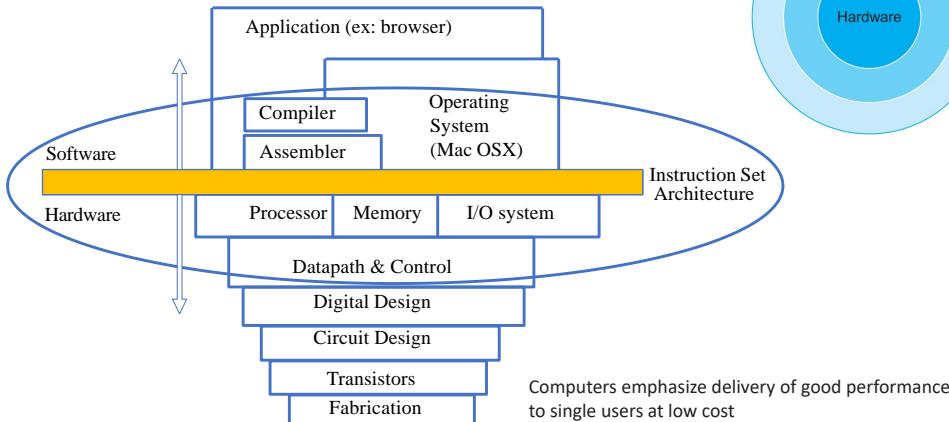


- Number of deployed devices continues to grow
 - Diversification of needs, architectures
 - Machine learning is common for most domains

8/28/2021

9

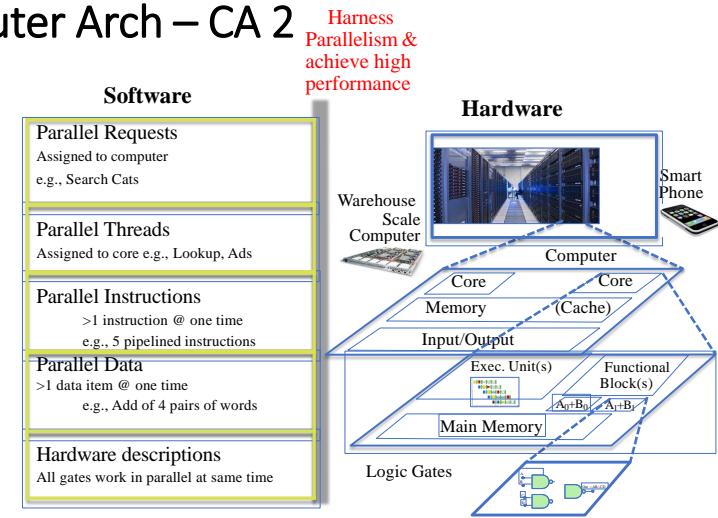
Thinking about machine structure in Computer Architecture 1



8/28/2021

10

Thinking about machine structure in Advanced Computer Arch – CA 2



8/28/2021

11

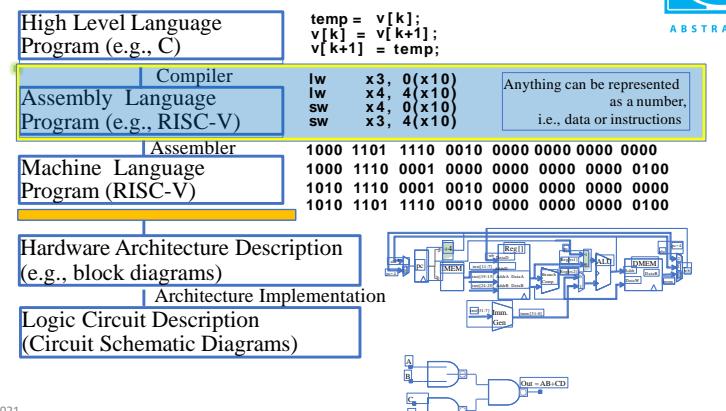
6 Great Ideas in Computer Architecture

- Abstraction (Layers of Representation/Interpretation)
- Moore's Law
- Principle of Locality/Memory Hierarchy
- Parallelism
- Performance Measurement & Improvement
- Dependability via Redundancy

8/28/2021

12

Great Idea #1: Abstraction (Layers of Representation/Interpretation)

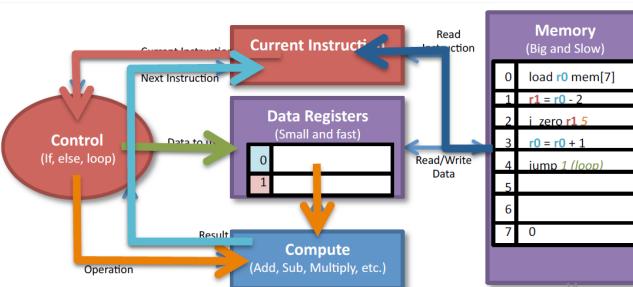


8/28/2021

13

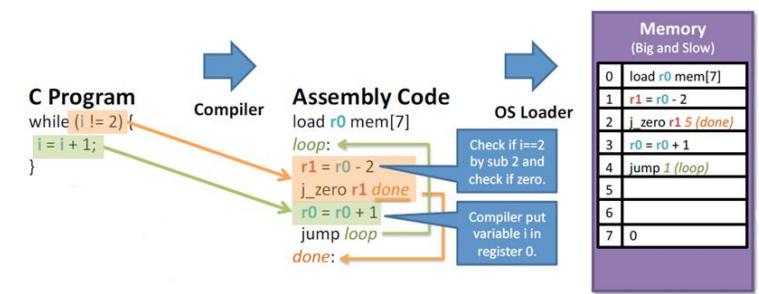
Below your program: Let's walk through the program

- What will the processor do?
- 1. Load the instruction
- 2. Figure out what operation to do
- 3. Figure out what data to use
- 4. Do the computation
- 5. Figure out next instruction
- Repeat this over and over and over...



8/28/2021

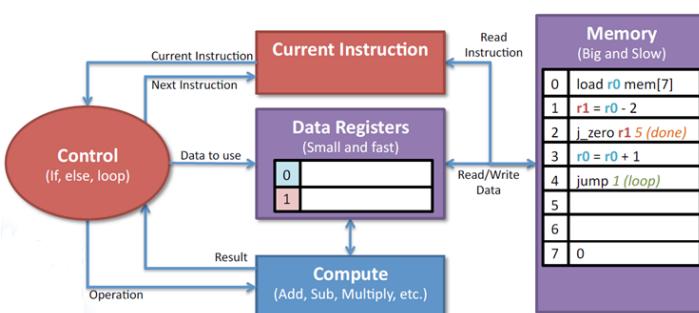
Example



8/28/2021

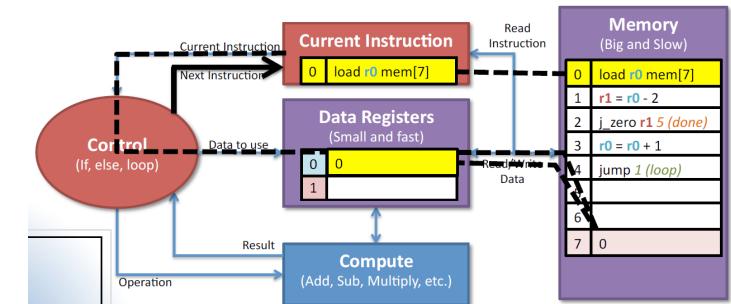
15

Basic processor: Memory, Control, and Compute



8/28/2021

1: Load r0 (i) from memory (location 7)

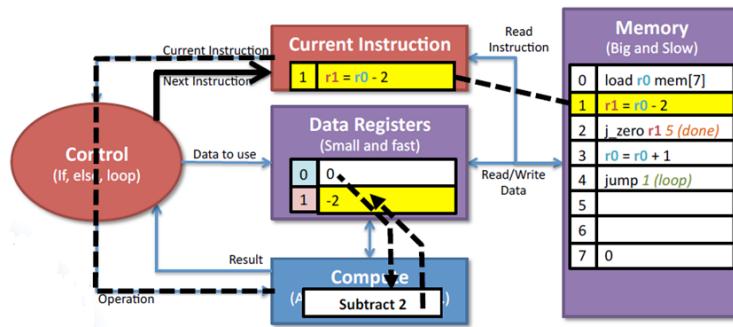


16

8/28/2021

17

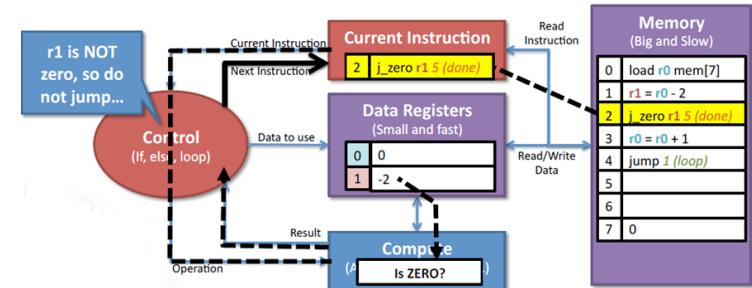
2: Subtract 2 from r0(i) to see if it is 2



8/28/2021

18

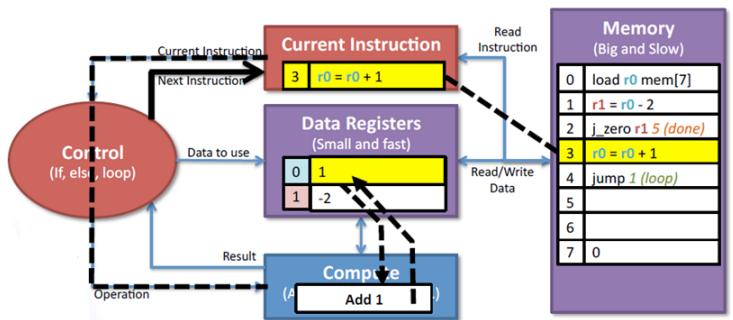
3: Check if r1 is zero 0, and jump to done if it is



8/28/2021

19

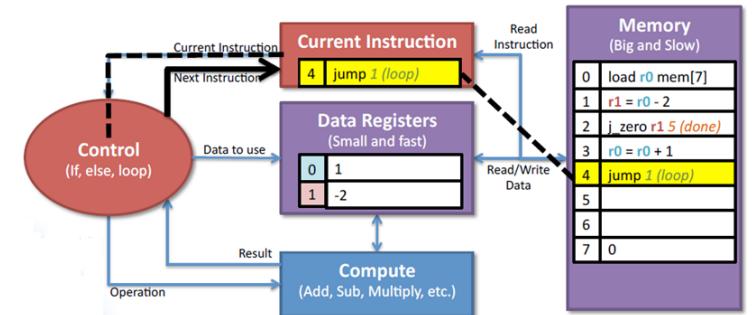
4: Increment r0 (i)



8/28/2021

20

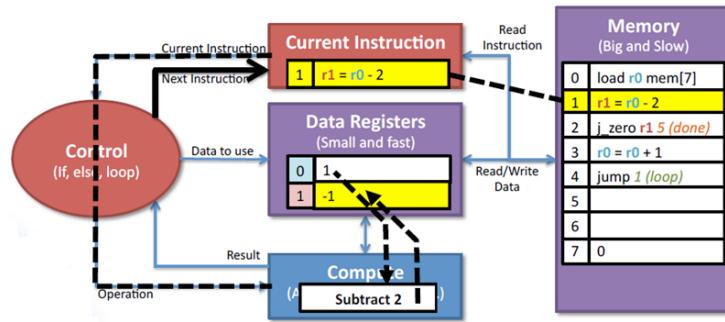
5: Continue the loop



8/28/2021

21

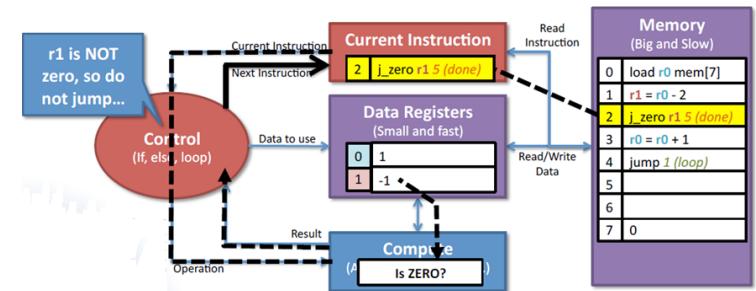
6: Subtract 2 from r0(i) to see if it is 2



8/28/2021

22

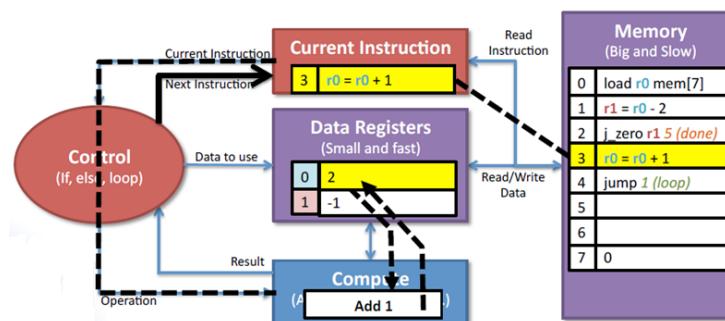
7: Check if r1 is zero 0, and jump to done if it is



8/28/2021

23

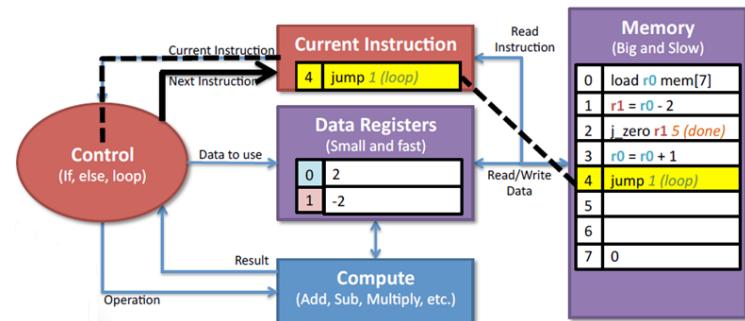
8: Increment r0 (i)



8/28/2021

24

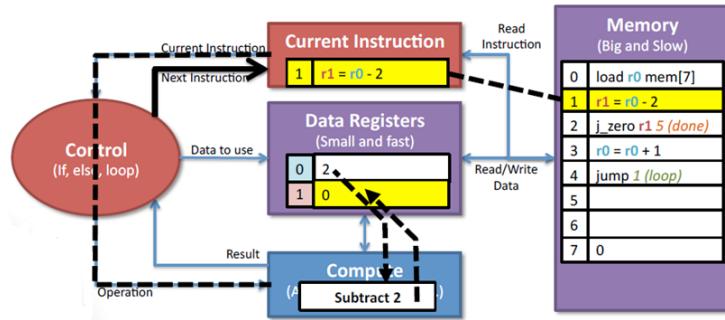
9: Continue the loop



8/28/2021

25

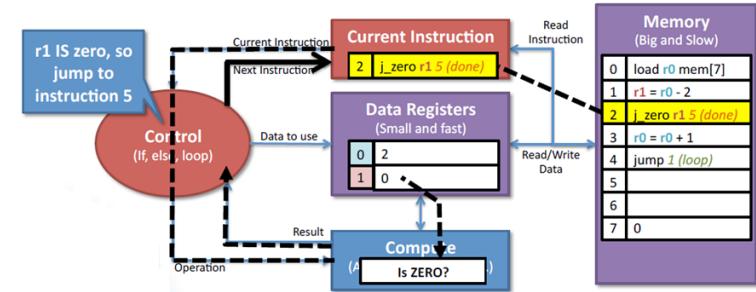
10: Subtract 2 from r0(i) to see if it is 2



8/28/2021

26

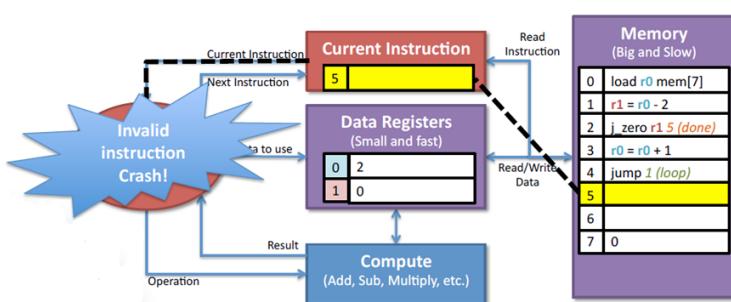
11: Check if r1 is zero 0, and jump to done if it is



8/28/2021

27

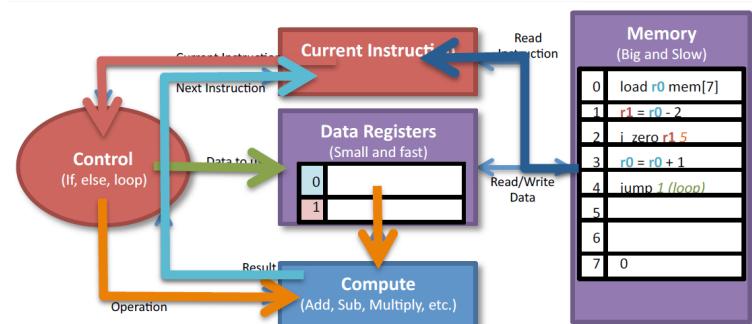
12: Crash because the instruction 5 is invalid!



8/28/2021

28

This course is about understanding the **details**, **corner cases**, **performance**, and how this all comes together in a RISC V processor.



8/28/2021

29

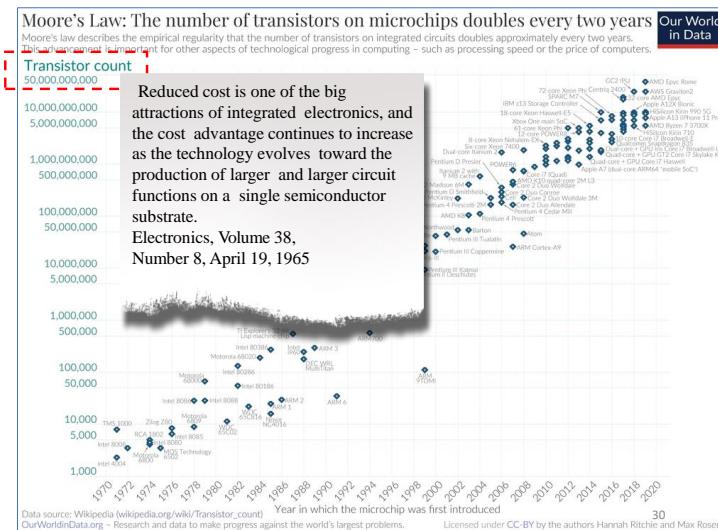
Great Idea #2: Moore's Law

It states that integrated circuit resources double every 18–24 months

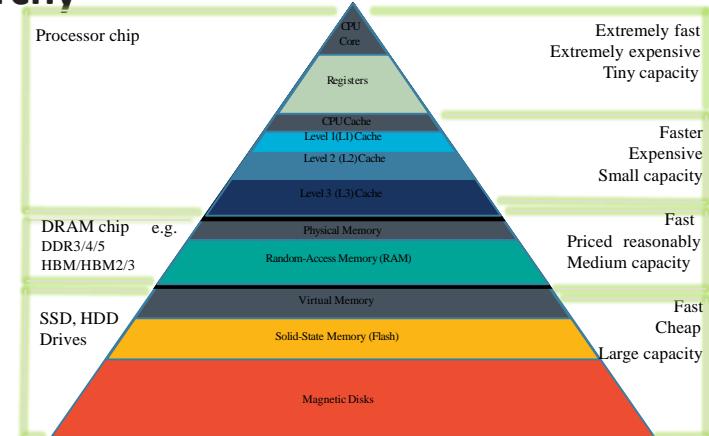


Gordon Moore
Intel Cofounder
B.S. Cal 1950!

8/28/2021



Great Idea #3 : Principle of Locality/Memory Hierarchy

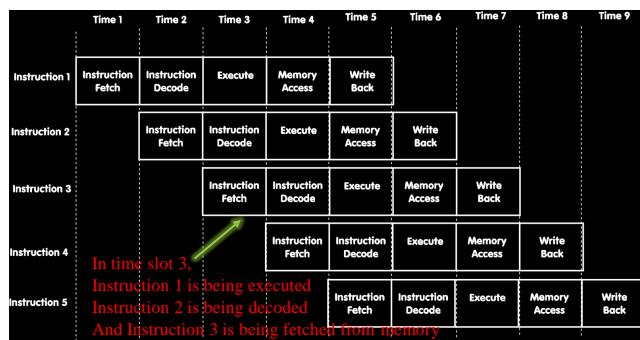


8/28/2021

31

Great Idea #4: Parallelism (1/3)

Performance via Pipelining

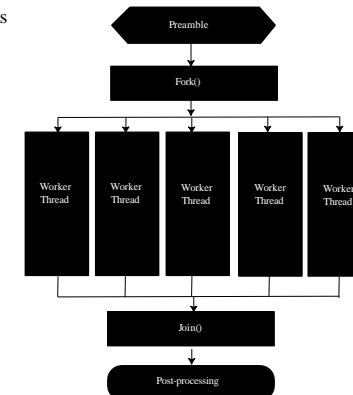


8/28/2021

32

Great Idea #4: Parallelism (2/3)

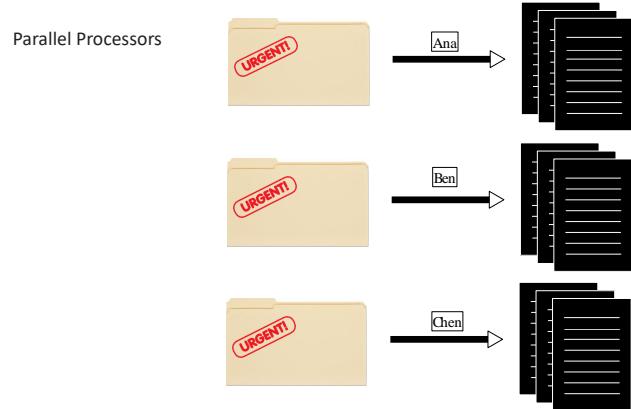
Parallel Threads



8/28/2021

33

Great Idea #4: Parallelism (3/3)



8/28/2021

34

Caveat! Amdahl's Law

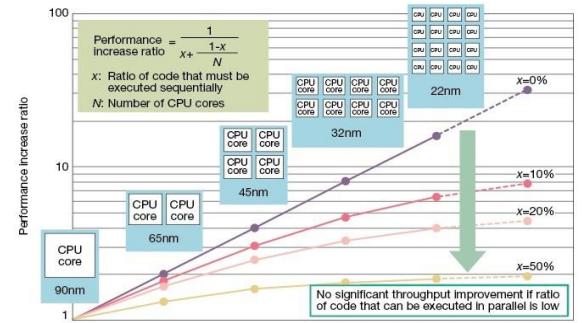


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.



Gene Amdahl
Computer Pioneer

8/28/2021

8/28/2021

35

Great Idea #5: Performance Measurement and Improvement

- How hardware and software affect performance ?

Hardware or software component	How this component affects performance
Algorithm	Determines both the number of source level statements and the number of I/O operations executed
Programming language, compiler, and architecture	Determines the number of computer instructions for each source level statement
Processor and memory system	Determines how fast instructions can be executed
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed

8/28/2021

36

Great Idea #5: Performance Measurement and Improvement

- Matching application to underlying hardware to exploit:
 - Locality
 - Parallelism
 - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency/Throughput
 - How long to set the problem up and complete it (or how many tasks can be completed in given time)
 - How much faster does it execute once it gets going
 - Latency is all about time to finish

8/28/2021

37

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail
- Applies to everything from datacenters to storage to memory to instructors
 - Redundant datacenters so that can lose 1 data center, but Internet service stays online
 - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
 - Redundant memory bits so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

8/28/2021

38

Homework

- Reading Technologies for Building Processors and Memory (Section 1.5 #page 24) and writing the report

8/28/2021

39

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolic, © 2020, UC Berkeley]

8/27/2021

1

8/27/2021

2

Instructions: Language of the Computer

- Introductions
- Operations of the Computer Hardware
- Operands of the Computer Hardware
- Logical Operations
- Instructions for Making Decisions

Introduction

- To demonstrate how easy it is to pick up other instruction sets, we will also take a quick look at two other popular instruction sets.
 - MIPS is an elegant example of the instruction sets designed since the 1980s. In several respects, RISC-V follows a similar design.
 - The Intel x86 originated in the 1970s, but still today powers both the PC and the Cloud of the post-PC era.

8/27/2021

3

Operations of the Computer Hardware

- Instruction set for a particular architecture (e.g. RISC-V) is represented by the Assembly language
- Each line of assembly code represents one instruction for the computer

Preliminary discussion of the logical design of an electronic computing instrument¹
Arthur W. Burks / Herman H. Goldstine / John von Neumann

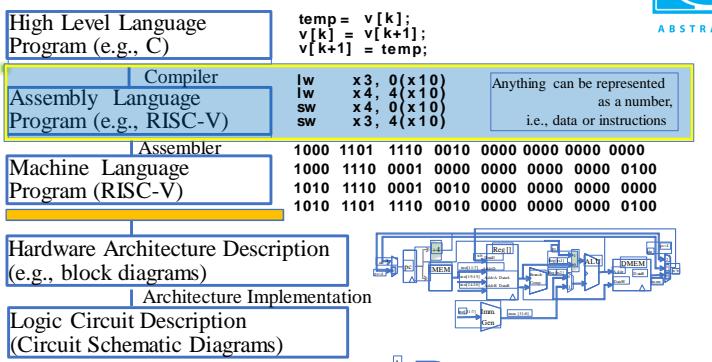
Instruction sets

3.1 It is easy to see by formal-logical methods that there exist codes that are *in abstracto* adequate to control and cause the execution of any sequence of operations which are individually available in the machine and which are in their entirety, conceivable by the problem planner. The really decisive considerations from the present point of view, in selecting a code, are more of a practical nature: simplicity of the equipment demanded by the code, and the clarity of its application to the actually important problems together with the speed of its handling of those problems. It would take us much too far afield to discuss these questions at all generally or from first principles. We will therefore restrict ourselves to analyzing only the type of code which we now envisage for our machine.

8/27/2021

4

Assembly Language



8/27/2021

5

Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
 - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence ...
- Different CPUs implement different sets of instructions. The set of instructions implemented in a particular CPU is an *Instruction Set Architecture (ISA)*.
 - Examples: ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM Power, IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, ...

8/27/2021

6

RISC-V Architecture

- New open-source, license-free ISA spec
 - Supported by growing shared software ecosystem
- Appropriate for all levels of computing system, from microcontrollers to supercomputers
- 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)
- Why RISC-V instead of Intel 80x86?
 - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
 - RISC-V has exponential adoption

8/27/2021

RISC V – GREEN CARD

RISC-V Reference Data	
ARITHMETIC CORE INSTRUCTION SET	
NAME	DESCRIPTION
aadd x5, x6, x7	Add with overflow detection
sub x5, x6, x7	Subtract
addi x5, x6, 20	Add immediate
ld x5, 40(x6)	Load doubleword
sd x5, 40(x6)	Store doubleword
lw x5, 40(x6)	Load word
sw x5, 40(x6)	Store word
lh x5, 40(x6)	Load halfword
sh x5, 40(x6)	Store halfword
lb x5, 40(x6)	Load byte
lbu x5, 40(x6)	Load byte, unsigned
st x5, x6	Store
lrd x5, x6	Load reserved
scd x5, x6	Store conditional
lui x5, 0x12345	Load upper immediate
andi x5, x6	And
ior x5, x6, x8	Inclusive or
xor x5, x6, x9	Exclusive or
and1 x5, x6, 20	And immediate
ori x5, x6, 20	Inclusive or/Immediate
xor1 x5, x6, 20	Exclusive or/Immediate
sll x5, x6, 7	Shift left logical
srl x5, x6, 7	Shift right logical
sra x5, x6, 7	Shift right arithmetic
sll1 x5, x6, 3	Shift left logical immediate
srl1 x5, x6, 3	Shift right logical immediate
srai x5, x6, 3	Shift right arithmetic immediate

<https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf>

<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

7

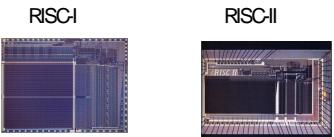
8/27/2021

8

RISC V Origin <https://cs61c.org/resources/>

- Started in Summer 2010 to support open research and teaching at UC Berkeley
 - Lineage can be traced to RISC-I/II projects (1980s)
 - As the project matured, it migrated to RISC-V foundation (www.riscv.org)
- Many commercial and research projects based on RISC-V, open-source and proprietary
- Widely used in education
- Read more:
 - <https://riscv.org/risc-v-history/>
 - <https://riscv.org/risc-v-genealogy/>

8/27/2021



RISC V Instructions

	Condition	Op	Op2	Op3	Op4	Op5	Op6	Op7	Op8	Op9	Op10	Op11	Op12	Op13	Op14	Op15	Op16	Op17	Op18	Op19	Op20	Op21	Op22	Op23	Op24	Op25	Op26	Op27	Op28	Op29	Op30	Op31		
Conditional branch		beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal																														
Conditional branch		bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal																														
Conditional branch		blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less																														
Conditional branch		bge x5, x6, 100	if (x5 > x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned																														
Unconditional branch		jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call																														
Unconditional branch		jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call																														

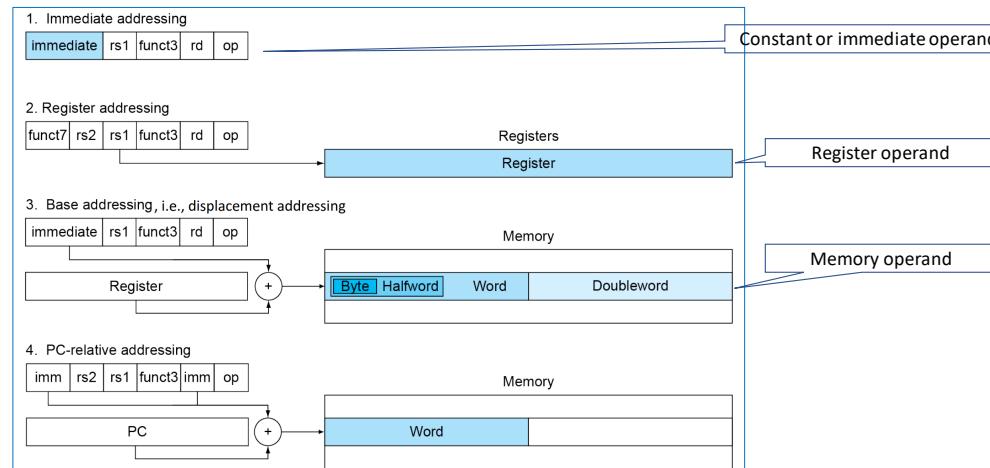
Arithmetic : +, /, *, -
 Logical: AND, OR, XOR
 Data transfer: Load, store
 Shift : >>, <<
 Condition branch: if ... then
 Unconditional branch: jump to loop

8/27/2021

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	$x5 = x6 + x7$	Three register operands; add
	Subtract	sub x5, x6, x7	$x5 = x6 - x7$	Three register operands; subtract
	Add immediate	addi x5, x6, 20	$x5 = x6 + 20$	Used to add constants
	Load doubleword	ld x5, 40(x6)	$x5 = \text{Memory}(x6 + 40)$	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	$\text{Memory}(x6 + 40) = x5$	Doubleword from register to memory
	Load word	lw x5, 40(x6)	$x5 = \text{Memory}(x6 + 40)$	Word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}(x6 + 40) = x5$	Word from register to memory
	Load halfword	lh x5, 40(x6)	$x5 = \text{Memory}(x6 + 40)$	Halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}(x6 + 40) = x5$	Unaligned halfword from memory to register
Data transfer	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}(x6 + 40)$	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}(x6 + 40)$	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}(x6 + 40) = x5$	Byte from register to memory
	Load reserved	lrd x5, x6	$x5 = \text{Memory}(x6)$	Load; 1st half of atomic swap
	Store conditional	scd x5, x6	$\text{Memory}(x6) = x5; x7 = 0$	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	$x5 = 0x12345000$	Loads 20-bit constant shifted left 12 bits
	And	and x5, x6, x7	$x5 = x6 \cdot x7$	Three reg. operands; bit-by-bit AND
	Inclusive or	ior x5, x6, x8	$x5 = x6 + x8$	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	$x5 = x6 \cdot x9$	Three reg. operands; bit-by-bit XOR
	And immediate	and1 x5, x6, 20	$x5 = x6 \cdot 20$	Bit-by-bit AND reg. with constant
	Inclusive or/Immediate	ori x5, x6, 20	$x5 = x6 + 20$	Bit-by-bit OR reg. with constant
	Exclusive or/Immediate	xor1 x5, x6, 20	$x5 = x6 \cdot 20$	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, 7	$x5 = x6 \ll 7$	Shift left by register
	Shift right logical	srl x5, x6, 7	$x5 = x6 \gg 7$	Shift right by register
	Shift right arithmetic	sra x5, x6, 7	$x5 = x6 \gg 7$	Arithmetic shift right by register
	Shift left logical immediate	sll1 x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srl1 x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	$x5 = x6 \gg 3$	Arithmetic shift right by immediate

10

Operand of the Computer Hardware: RISC-V Addressing Summary



8/27/2021

11

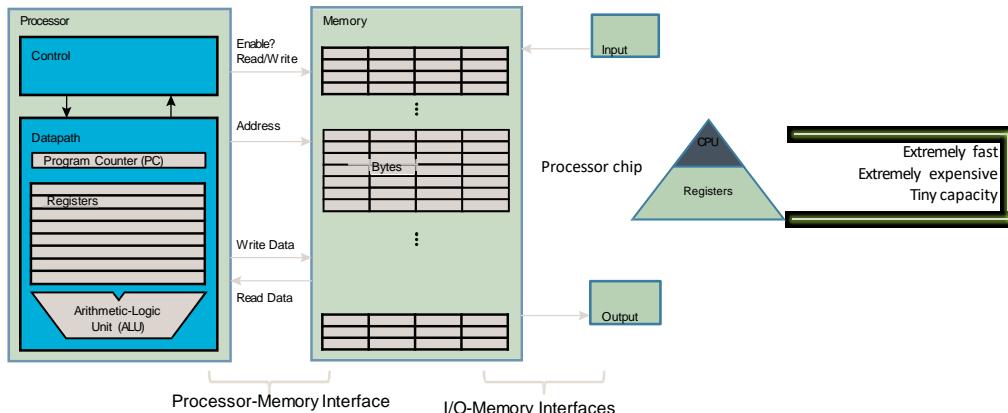
Assembly Variables: Registers (1/3)

- Elements of hardware: registers
- Unlike HLL like C or Java, assembly cannot use variables
 - Why not? Keep Hardware Simple
- Assembly operands are registers
 - Limited number of special locations built directly into the hardware
 - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they're very fast (faster than 0.25ns)
 - Recall light is $3 \times 10^8 \text{m/s} = 0.3 \text{m/ns} = 30 \text{cm/ns} = 10 \text{cm}/0.3\text{ns}!!...$ where 0.3ns is the clock period of a 3.33GHz computer

8/27/2021

12

Aside: Registers are Inside the Processor



8/27/2021

13

Assembly Variables: Registers (2/3)

- Drawback: Since registers are in hardware, there is a predetermined number of them
 - Solution: RISC-V code must be very carefully put together to efficiently use registers
- 32 registers in RISC-V. Why 32?
 - Smaller is faster, but too small is bad. Goldilocks principle ("This porridge is too hot; This porridge is too cold; this porridge is just right")
- Each RISC-V register is 32 bits wide (in RV32 variant)
 - Groups of 32 bits called a word in RV32
 - P&H textbook uses the 64-bit variant RV64

8/27/2021

14

Assembly Variables: Registers (3/3)

- Registers are numbered from 0 to 31
 - Referred to by number x0–x31
- X0 is special, always holds value zero
 - So only 31 registers able to hold variable values
- Each register can be referred to by number or name
 - Will add names later

8/27/2021

Register	ABI Name	Description	Saver
x0	zero	hardwired zero	-
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5-7	t0-2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-11	a0-1	function arguments / return values	Caller
x12-17	a2-7	function arguments	Caller
x18-27	s2-11	saved registers	Callee
x28-31	t3-6	temporary registers	15 Caller

C, Java variables vs. registers

- In C (and most high-level languages) variables declared first and given a type. E.g.,

```
int fahr, celsius;
char a, b, c, d, e;
```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In assembly language, the registers have no type
 - Operation determines how register contents are treated

8/27/2021

16

Comments in Assembly

- Make your code more readable: comments!
 - Hash (#) is used for RISC-V comments anything from hash mark to end of line is a comment and will be ignored
- This is just like the C99 //
Note: Different from C.
- C comments have format /* comment */ so they can span many lines
- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
 - Unlike in C (and most other high-level languages), each line of assembly code contains at most 1 instruction
 - Instructions are related to operations (=, +, -, *, /) in C or Java

8/27/2021

17

RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:

```
one two, three, four
add x1, x2, x3
```
- where:

```
one = operation by name
two = operand getting result ("destination," x1)
three = 1st operand for operation ("source1," x2)
four = 2nd operand for operation ("source2," x3)
```
- Syntax is rigid: 1 operator, 3 operands
- Why? Keep hardware simple via regularity

8/27/2021

18

Addition and Subtraction of Integers (2/4)

- Addition in Assembly

- Example: `add x1, x2, x3 (in RISC-V)`
- Equivalent to: $a = b + c$ (in C)
- where C variables \Leftrightarrow RISC-V registers are: $a \Leftrightarrow x1$, $b \Leftrightarrow x2$, $c \Leftrightarrow x3$

- Subtraction in Assembly

- Example: `sub x3, x4, x5 (in RISC-V)`
- Equivalent to: $d = e - f$ (in C)
- where C variables \Leftrightarrow RISC-V registers are:
 $d \Leftrightarrow x3$, $e \Leftrightarrow x4$, $f \Leftrightarrow x5$

8/27/2021

19

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?

$a = b + c + d - e;$

- Break into multiple instructions

```
add x10, x1, x2 # a_temp = b + c  
add x10, x10, x3 # a_temp = a_temp + d  
sub x10, x10, x4 # a = a_temp - e
```

- Notice: A single line of C may break up into several lines of RISC-V.

- Notice: Everything after the hash mark on each line is ignored (comments).

8/27/2021

8/27/2021

20

Addition and Subtraction of Integers (4/4)

- How do we do this?

$f = (g + h) - (i + j);$

- Use intermediate temporary register

```
add x5, x20, x21 # a_temp = g + h  
add x6, x22, x23 # b_temp = i + j  
sub x19, x5, x6 # f = (g + h) - (i + j)
```

8/27/2021

21

Immediates

- There is no Subtract Immediate in RISC-V: Why? There are **add** and **sub**, but no **addi** counterpart

- Limit types of operations that can be done to absolute minimum if an operation can be decomposed into a simpler

- where RISC-V registers **x3, x4** are associated with C variables f, g, respectively

```
addi x3, x4, -10 (in RISC-V)  
f = g - 10 in C
```

8/27/2021

22

Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So the register zero (x0) is 'hard-wired' to value 0; e.g.

```
add x3,x4,x0 # in RISC V  
f = g # in C
```

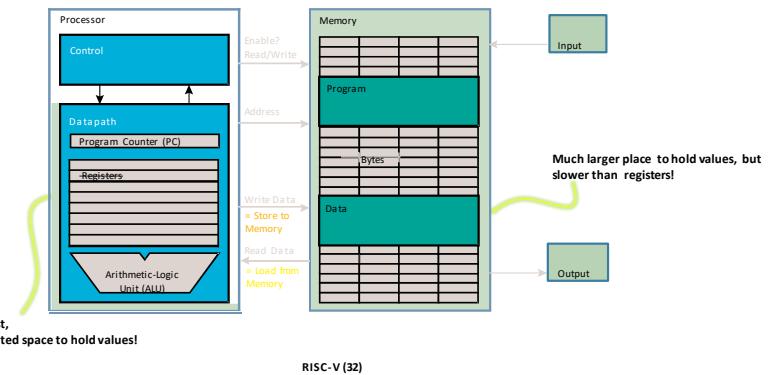
- where RISC-V registers **x3, x4** are associated with C variables **f, g**
- Defined in hardware, so an instruction

```
add x0,x3,x4 # will not do anything!
```

8/27/2021

23

Data Transfer: Load from and Store to memory



8/27/2021

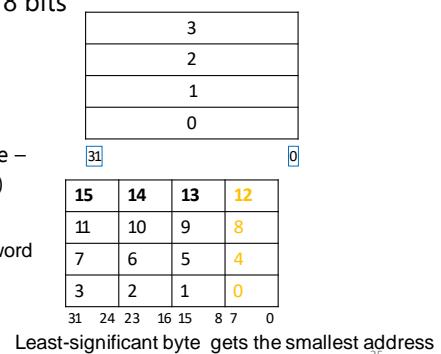
24

Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)—works fine if everything is a multiple of 8 bits
 - 8 bit chunk is called a byte (1 word = 4 bytes)
 - Memory addresses are really in bytes, not words

- Word addresses are 4 bytes apart
 - Word address is same as address of rightmost byte – least-significant byte (i.e. Little-endian convention)

Least-significant byte in a word



8/27/2021

25

Homework_ week 2

- Exercise 2.1 – 2.11 (pg. 162 -164 in P&H textbook)

26

Big Endian vs. Little Endian

The adjective endian has its origin in the writings of 18th century writer Jonathan Swift. In the 1726 novel Gulliver's Travels, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end. He called them the "Big-Endians" and the "Little-Endians".

The order in which BYTES are stored in memory
Bits always stored as usual (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we typically write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Big Endian Little Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
00000000	00000000	00000000	00000000

DRAM chip e.g. DDR3/4/5 HBM/HBM2/3

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000000	00000000

Physical Memory Random -Access Memo ry (RAM)

00000000	00000000	00000100	00000001
----------	----------	----------	----------

Examples Names in China or Hungary (e.g., Nikolic Bora)

Java Packages: (e.g., org.mypackage.HelloWorld)

Dates in ISO 8601 YYYY-MM-DD (e.g., 2020-09-07)

Eating Pizza crust first

Examples Names in the US (e.g., BoraNikolic)

Internet names (e.g., cs.berkeley.edu)

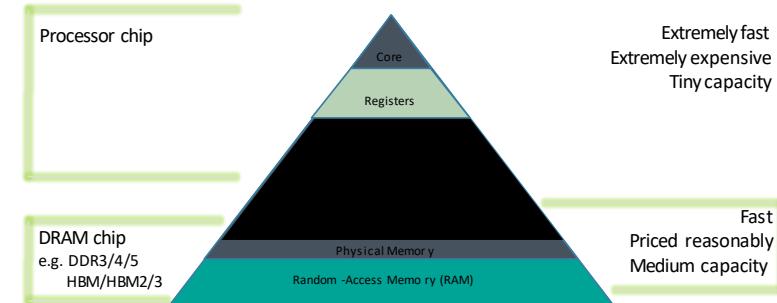
Dates written in Europe DD/MM/YYYY (e.g., 07/09/2020)

Eating Pizza skinny part first

8/27/2021

27

Speed of Registers vs. Memory



- Registers: 32 words (128 Bytes)
- Memory (DRAM): Billions of bytes (2 GB to 64 GB on laptop) and physics dictates that **Smaller is Faster**
- How much faster are registers than DRAM??
 - About 50-500 times faster! (in terms of latency of one access - tens of ns)
 - But subsequent words come every few ns

28

Load from Memory to Register

- C code

```
int A[100]; g = h + A[3];
```



- Using Load Word (lw) in RISC-V:

```
lw x10,12(x15) # Reg x10 gets A[3]
add x11,x12,x10 # g = h + A[3]
# x15 - base register (pointer to A[0]) 12 - offset in bytes
# Offset must be a constant known at assembly time
```

8/27/2021

29

Store from Register to Memory

- C code

```
int A[100]; A[10] = h + A[3];
```



- Using Store Word (sw) in RISC-V:

```
lw x10,12(x15) # Temp reg x10 gets A[3]
add x10,x12,x10 # Temp reg x10 gets h + A[3]
sw x10,40(x15) # A[10] = h + A[3]
```

- Note:

x15 - base register (pointer) 12,40 - offsets in bytes
x15+12 and x15+40 must be multiples of 4

30

8/27/2021

Loading and Storing Bytes

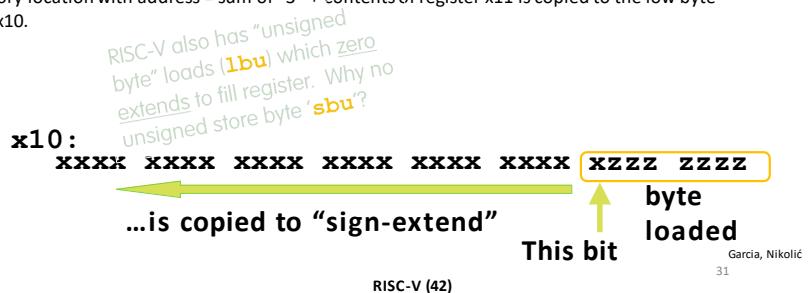
- In addition to word data transfers (lw, sw), RISC-V has byte data transfers:

load byte: lb
store byte: sb

- Same format as lw, sw. E.g.,

lb x10,3(x11)

→ contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.



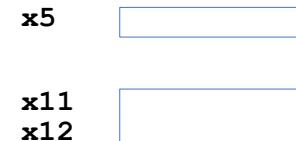
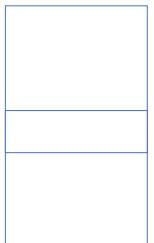
8/27/2021

31

Example: What is in x12 ?

```
addi x11,x0,0x3F5  
sw x11,0(x5)  
lb x12,1(x5)
```

Memory



8/27/2021

Garcia, Nikolic
32

Substituting addi

- The following two instructions:

```
lw x10,12(x15) # Temp reg x10 gets A[3]  
add x12,x12,x10 # reg x12 = reg x12 + A[3]
```

- Replace addi:

```
addi x12, value # value in A[3]
```

- But involve a load from memory!

- Add immediate is so common that it deserves its own instruction!

RV32 So Far...

- Addition/subtraction

```
add rd, rs1, rs2 sub rd, rs1, rs2
```

- Add immediate

```
addi rd, rs1, imm
```

- Load/store

```
lw rd, rs1, imm
```

```
lb rd, rs1, imm
```

```
lbu rd, rs1, imm
```

```
sw rs1, rs2, imm
```

```
sb rs1, rs2, imm
```

8/27/2021

33

8/27/2021

34

Computer Decision Making

- Based on computation, do something different
- In programming languages: if-statement
- RISC-V: if-statement instruction is
beq reg1, reg2, L1
- means: go to statement labeled L1 if (value in reg1) == (value in reg2)otherwise, go to next statement
- beq stands for branch if equal
- Other instruction: bne for branch if not equal

8/27/2021

35

Types of Branches

- Branch – change of control flow
- Conditional Branch – change control flow depending on outcome of comparison
 - branch if equal (**beq**) or branch if not equal (**bne**)
 - Also branch if less than (**blt**) and branch if greater than or equal (**bge**)
 - And unsigned versions (**bltu**, **bgeu**)
- Unconditional Branch – always branch
 - a RISC-V instruction for this: *jump* (**j**), as in
j label

8/27/2021

36

Example *if* Statement

- Assuming translations below, compile if block

$f \rightarrow x10$ $g \rightarrow x11$ $h \rightarrow x12$
 $i \rightarrow x13$ $j \rightarrow x14$

if (i == j)
 f = g + h;

bne x13,x14,Exit
 add x10,x11,x12
Exit:

- May need to negate branch condition

8/27/2021

37

Example *if-else* Statement

- Assuming translations below, compile

$f \rightarrow x10$ $g \rightarrow x11$ $h \rightarrow x12$
 $i \rightarrow x13$ $j \rightarrow x14$

if (i == j)
 f = g + h;
Else
 f = g - h;

bne x13,x14,Else
 add x10,x11,x12
j Exit
Else:sub x10,x11,x12
Exit:

38

Magnitude Compares in RISC-V

- General programs need to test < and > as well.

- RISC-V magnitude-compare branches:

“Branch on Less Than”

Syntax: `blt reg1,reg2, Label`

Meaning: `if (reg1 < reg2) goto Label;`

“Branch on Less Than Unsigned”

Syntax: `bltu reg1,reg2, Label`

Meaning: `if (reg1 < reg2) // treat registers
as unsigned integers
goto label;`

Also “Branch on Greater or Equal” `bge` and `bgeu`

Note: No ‘`bgt`’ or ‘`ble`’ instructions

8/27/2021

39

Loops in C/Assembly

- There are three types of loops in C:
- while
- do ... while
- for
- Each can be rewritten as either of the other two, so the same branching method can be applied to these loops as well.
- Key concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision-making is conditional branch

8/27/2021

40

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
sum += A[i];
Assume that the base address of A, sum
and i correspond to registers x9,x10
and x11
```

```
add x9, x8, x0 # x9=&A[0]
add x10, x0, x0 # sum
add x11, x0, x0 # i
addi x13,x0, 20 # x13
Loop:
bge x11,x13,Done
lw x12, 0(x9)    # x12 A[i]
add x10,x10,x12 # sum+=
addi x9, x9,4    # &A[i+1]
addi x11,x11,1   # i++
j Loop
Done:
```

8/27/2021

41

C Loop Mapped to RISC-V Assembly

Here is a traditional loop in C:
`while (save[i] == k)
i += 1;`
Assume that i and k correspond to registers x22 and x24 and the base
of the array save is in x25. What is the RISC-V assembly code
corresponding to this C code?

8/27/2021

42

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great Ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolic, © 2020, UC Berkeley]

8/27/2021

1

RV32 So Far...

- Add/sub
 - add rd, rs1, rs2
 - sub rd, rs1, rs2
- Add immediate
 - addi rd, rs1, imm
- Load/store
 - lw lb
 - lbu rd,
 - sw rs1, rs2, imm
 - sb rs1, rs2, imm
- Branching
 - beq rs1, rs2, Label
 - bne rs1, rs2, Label
 - bge rs1, rs2, Label
 - blt rs1, rs2, Label
 - bgeu rs1, rs2, Label
 - bitu rs1, rs2, Label
 - j Label

8/27/2021

2

RISC-V Logical Instructions

- Useful to operate on fields of bits within a word
 - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called logical operations

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	and
Bit-by-bit OR			or
Bit-by-bit XOR	^	^	xor
Shift left logical	<<	<<	sll
Shift right logical	>>	>>	srl

8/27/2021

8/27/2021

RISC-V Logical Instructions

- Always two variants
 - Register: **and x5, x6, x7 # x5 = x6 & x7**
 - Immediate: **andi x5, x6, 3 # x5 = x6 & 3**
- Used for ‘masks’
 - **andi** with **0000 00FF_{hex}** isolates the least significant byte
 - **andi** with **FF00 0000_{hex}** isolates the most significant byte
- There is no logical NOT in RISC-V
 - Use **xor** with **11111111_{two}**
 - Remember - simplicity...

4

Logical Shifting

- Shift Left Logical (**slli**) and immediate (**slli**):

slli x11, x12, 2 #x11=x12<<2

- Store in **x11** the value from **x12** shifted by 2 bits to the left (they fall off end), inserting 0's on right; **<<** in C.

- Before: **0000 0002_{hex}**

0000 0000 0000 0000 0000 0000 0000 0010_{two}

- After: **0000 0008_{hex}**

0000 0000 0000 0000 0000 0000 0000 1000_{two}

- What arithmetic effect does shift left have?

- Shift Right: **srl** is opposite **shift**; **>>**

8/27/2021

5

8/27/2021

6

Arithmetic Shifting

- Shift right arithmetic (**sra**, **srai**) moves n bits to the right (insert high-order sign bit into empty bits)

- For example, if register **x10** contained

1111 1111 1111 1111 1111 1111 1110 0111_{two} = -25_{ten}

- If execute **srai x10, x10, 4**, result is:

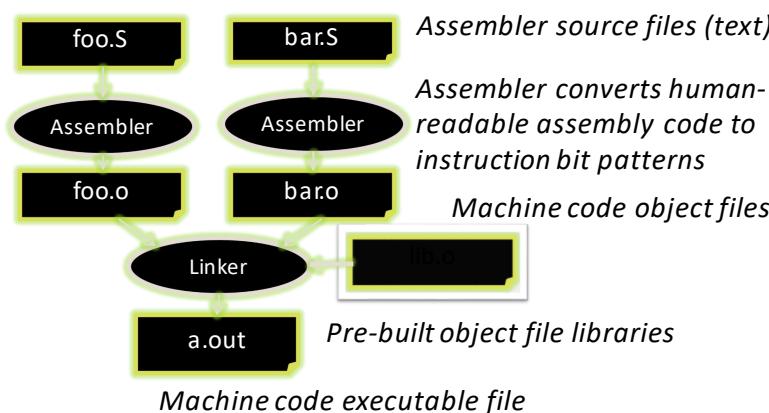
1111 1111 1111 1111 1111 1111 1111 1110_{two} = -2_{ten}

- Unfortunately, this is NOT same as dividing by 2^n

- Fails for odd negative numbers

- C arithmetic semantics is that division should round towards 0

Assembler to Machine Code (More Later in Course)



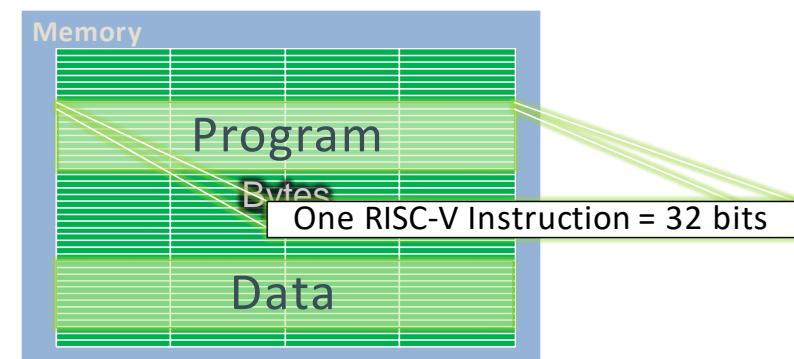
8/27/2021

7

8/27/2021

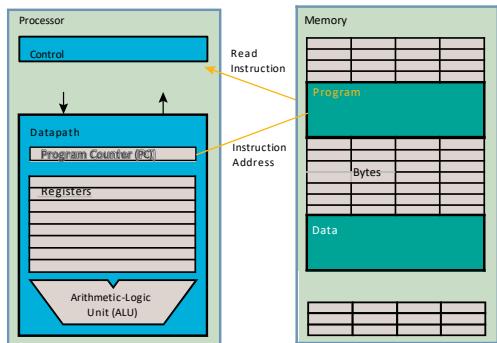
8

How Program is Stored



RISC-V

Program Execution



■ PC (program counter) is a register internal to the processor that holds byte address of next instruction to be executed

Helpful RISC-V Assembler Features

- Symbolic register names
 - E.g., **a0-a7** for argument registers (**x10-x17**) for function calls
 - E.g., **zero** for **x0**
- Pseudo-instructions
 - Shorthand syntax for common assembly idioms
 - E.g., **mv rd, rs** = **addi rd, rs, 0**
 - E.g., **li rd, 13** = **addi rd, x0, 13**
 - E.g., **nop** = **addi x0, x0, 0**

RISC V Function calls

```
main() {
    int i,j,k,m;
    ...
    i = mult(j,k); ...
    m = mult(i,i); ...
    /* really dumb mult function */
    int mult (int mcand, int mlier){
        int product = 0;
        while (mlier > 0) {           What instructions can
            product = product + mcand;  accomplish this?
            mlier = mlier -1; }
        return product;
    }
```

What information must compiler/programmer keep track of?

Six Fundamental Steps in Calling a Function

1. Put arguments in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put return value in a place where calling code can access it and restore any registers you used; release local storage
6. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- a0-a7 (x10-x17)**: eight *argument* registers to pass parameters and two return values (**a0-a1**)
- ra**: one *return address* register to return to the point of origin (**x1**)
- Also **s0-s1 (x8-x9)** and **s2-s11 (x18-x27)**: saved registers (more about those later)

8/27/2021

13

Instruction Support for Functions (1/4)

```
... sum(a,b);... /* a,b:s0,s1 */  
}  
int sum(int x, int y) {  
    return x+y;  
}  
  
address (shown in decimal)  
1000  
1004  
1008  
1012  
1016  
...  
2000  
2004
```

RISC-V

In RISC-V, all instructions are 4 bytes, and stored in memory just like data. So, here we show the addresses of where the programs are stored.

14

Instruction Support for Functions (2/4)

```
... sum(a,b);... /* a,b:s0,s1 */  
}  
int sum(int x, int y) {  
    return x+y;  
}  
  
address (shown in decimal)  
1000 mv a0,s0      # x = a  
1004 mv a1,s1      # y = b  
1008 addi ra,zero,1016 #ra=1016  
1012 j   sum        #jump to sum  
1016 ...           # next inst.  
...  
2000 sum: add a0,a0,a1  
2004 jr  ra #new instr. "jump reg"
```

C

RISC-V

8/27/2021

15

Instruction Support for Functions (3/4)

```
... sum(a,b);... /* a,b:s0,s1 */  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

C

RISC-V

8/27/2021

- Question: Why use **jr** here? Why not use **j**?
- Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

```
...  
2000 sum: add a0,a0,a1  
2004 jr  ra #new instr. "jump reg"
```



16

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address:
jump and link (**jal**)

- Before:

```
1008 addi ra,zero,1016      # ra=1016
1012 j sum                 # goto sum
```

- After

```
1008 jal sum   # ra=1012,goto sum
```

- Why have a **jal**?

- Make the common case fast: function calls very common
- Reduce program size

Don't have to know where code is in memory with **jal**!

8/27/2021

17

8/27/2021

18

RISC-V Function Call Instructions

- Invoke function: *jump and link* instruction (**jal**) (really should be **laj** “link and jump”)
 - “link” means form an *address* or *link* that points to calling site to allow function to return to proper address
 - Jumps to address and simultaneously saves the address of the following instruction in register **ra**
- **jal FunctionLabel**
- Return from function: *jump register* instruction (**jr**)
 - Unconditional jump to address specified in register: **jr ra**
 - Assembler shorthand: **ret = jr ra**

Summary of Instruction Support

Actually, only two instructions:

- **jal rd, Label** – jump-and-link
- **jalr rd, rs, imm** – jump-and-link register

j, jr and **ret** are pseudoinstructions!

- **j: jal x0, Label**

8/27/2021

19

8/27/2021

20

RISC-V Instruction Representation

High Level Language
Program (e.g., C)

Assembly Language
Program (e.g., RISC-V)

Machine Language
Program (RISC-V)

Hardware Architecture Description
(e.g., block diagrams)

Logic Circuit Description
(Circuit Schematic Diagrams)

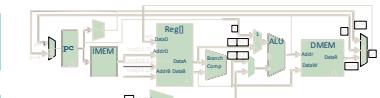
Anything can be represented
as a number,
i.e., data or instructions

lw x3, 0(x10)
lw x4, 4(x10)
sw x4, 0(x10)
sw x3, 4(x10)

Machine Language Program (RISC-V)

Hardware Architecture Description
(e.g., block diagrams)

Logic Circuit Description
(Circuit Schematic Diagrams)



Instructions as Numbers(1/2)

- Most data we work with is in words (32-bit chunks):
 - Each register is a word
 - **lw** and **sw** both access memory one word at a time
- So how do we represent instructions?
 - Remember: Computer only understands 1s and 0s, so assembler string “**add x10, x11, x0**” is meaningless to hardware
 - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
 - Same 32-bit instructions used for RV32, RV64, RV128

8/27/2021

21

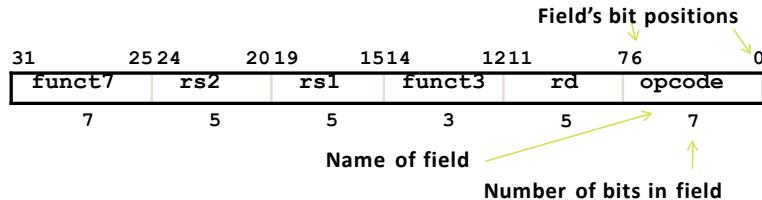
Instructions as Numbers(2/2)

- One word is 32 bits, so divide instruction word into “**fields**”
- Each field tells processor something about instruction
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define six basic types of instruction formats:
 - R-format for register-register arithmetic operations
 - I-format for register-immediate arithmetic operations and loads
 - S-format for stores
 - B-format for branches (minor variant of S-format)
 - U-format for 20-bit upper immediate instructions
 - J-format for jumps (minor variant of U-format)

8/27/2021

22

R-Format Instruction Layout

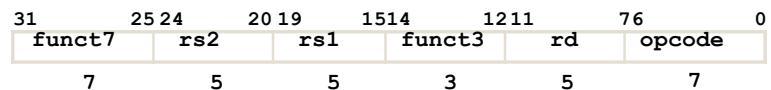


- 32-bit instruction word divided into six fields of varying numbers of bits each: $7+5+5+3+5+7 = 32$
- Examples
 - **opcode** is a 7-bit field that lives in bits 6-0 of the instruction
 - **rs2** is a 5-bit field that lives in bits 24-20 of the instruction

8/27/2021

23

R-Format Instructions opcode/funct Fields



opcode: partially specifies what instruction it is

- Note: This field is equal to 0110011_{two} for all R-Format register-register arithmetic instructions

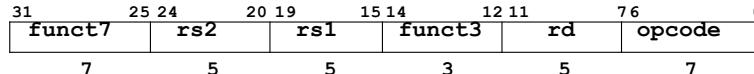
funct7+funct3: combined with **opcode**, these two fields describe what operation to perform

- **Question: You have been professing simplicity, so why aren't opcode and funct7 and funct3 a single 17-bit field?**
 - We'll answer this later

8/27/2021

24

R-Format Instructions Register Specifiers



rs1 (Source Register #1): specifies register containing first operand

rs2: specifies second register operand

rd (Destination Register): specifies register which will receive result of computation

Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**x0-x31**)

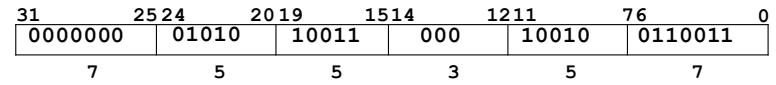
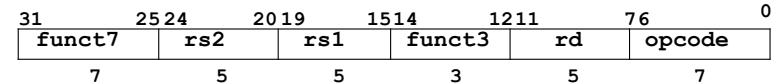
8/27/2021

25

R-Format Example

- RISC-V Assembly Instruction:

add x18,x19,x10



add rs2=10 rs1=19 add rd=18 Reg-Reg OP

8/27/2021

26

Your Turn

- What is correct encoding of **add x4, x3, x2** ?

- 1) 40218233_{hex}
- 2) 002182b3_{hex}
- 3) 402182b3_{hex}
- 4) 00218233_{hex}
- 5) 00218234_{hex}

31 25 24 20 19 15 14 12 11 7 6 0

0000000	rs2	rs1	000	rd	0110011
0100000	rs2	rs1	000	rd	0110011
0000000	rs2	rs1	100	rd	0110011
0000000	rs2	rs1	110	rd	0110011

add
sub
xor
or
and

8/27/2021

27

All RV32 R-format Instructions

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

Different encoding in funct7 + funct3 selects different operations
Can you spot two new instructions?

8/27/2021

28

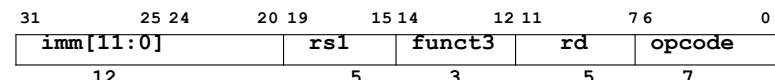
I-Format Instructions

- What about instructions with immediates?
 - Compare:
 - `add rd, rs1, rs2`
 - `addi rd, rs1, imm`
 - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
 - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is mostly consistent with R-format
 - Notice if instruction has immediate, then uses at most 2 registers (one source, one destination)

8/27/2021

29

I-Format Instruction Layout



- Only one field is different from R-format, `rs2` and `funct7` replaced by 12-bit signed immediate, `imm[11:0]`
- Remaining fields (`rs1, funct3, rd, opcode`) same as before
- `imm[11:0]` can hold values in range $[-2048_{\text{ten}}, +2047_{\text{ten}}]$
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- We'll later see how to handle immediates > 12 bits

8/27/2021

30

I-Format Example

- RISC-V Assembly Instruction:
`addi x15, x1, -50`

31	20 19	15 14	12 11	7 6	0
<code>imm[11:0]</code>	<code>rs1</code>	<code>funct3</code>	<code>rd</code>	<code>opcode</code>	
12	5	3	5	7	
<code>1111111001110</code>	<code>00001</code>	<code>000</code>	<code>01111</code>	<code>0010011</code>	
<code>imm=-50</code>	<code>rs1=1</code>	<code>add</code>	<code>rd=15</code>	<code>OP-Imm</code>	

8/27/2021

31

All RV32 I-format Arithmetic Instructions

<code>imm[11:0]</code>	<code>rs1</code>	<code>000</code>	<code>rd</code>	<code>0010011</code>	<code>addi</code>
<code>imm[11:0]</code>	<code>rs1</code>	<code>010</code>	<code>rd</code>	<code>0010011</code>	<code>slti</code>
<code>imm[11:0]</code>	<code>rs1</code>	<code>011</code>	<code>rd</code>	<code>0010011</code>	<code>sltiu</code>
<code>imm[11:0]</code>	<code>rs1</code>	<code>100</code>	<code>rd</code>	<code>0010011</code>	<code>xori</code>
<code>imm[11:0]</code>	<code>rs1</code>	<code>110</code>	<code>rd</code>	<code>0010011</code>	<code>ori</code>
<code>imm[11:0]</code>	<code>rs1</code>	<code>111</code>	<code>rd</code>	<code>0010011</code>	<code>andi</code>
<code>0000000</code>	<code>shamt</code>	<code>rs1</code>	<code>001</code>	<code>rd</code>	<code>slli</code>
<code>0000000</code>	<code>shamt</code>	<code>rs1</code>	<code>101</code>	<code>rd</code>	<code>srlti</code>
<code>0100000</code>	<code>shamt</code>	<code>rs1</code>	<code>101</code>	<code>rd</code>	<code>srai</code>

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bitpositions)

32

Xác định mã máy (Hệ hex) cho các lệnh sau

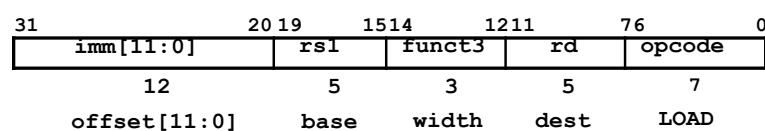
- add x1, x2, x3 : 0x003100B3
- addi x8, x6, -10: 0xFF630413
- ori x7, x10, 0x04: 0x00456393
- slli x5, x12, 2: 0x00261293

0000 000	0 0011	0001 0	000	0000 1	011 0011	add
1111 1111 0110	0011 0	000	0100 0	001 0011	addi	
0000 0000 0100	0101 0	110	0011 1	001 0011	ori	
0000 000	0 0010	0110 0	001	0010 1	001 0011	slli

8/27/2021

33

Load Instructions are also I-Type



- The 12-bit signed immediate is added to the base address in register **rs1** to form the memory address
 - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register **rd**

8/27/2021

34

I-Format Load Example

- RISC-V Assembly Instruction:

lw x14, 8(x2)

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
offset[11:0]	base	width	dest	LOAD	
000000001000	00010	010	01110	0000011	
imm=+8	rs1=2	lw	rd=14	LOAD	
(load word)					

8/27/2021

35

All RV32 Load Instructions

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

funct3 field encodes size and
'signedness' of load data

- lbu is “load unsigned byte”
- lh is “load halfword”, which loads 16 bits (2 bytes) and sign- extends to fill destination 32-bit register
- lhu is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no ‘lwu’ in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

8/27/2021

36

S-Format Used for Stores

31 Imm[11:5]	25-24 rs2	20-19 rs1	15-14 funct3	12-11 imm[4:0]	7-6 opcode	0
-----------------	--------------	--------------	-----------------	-------------------	---------------	---

offset[11:5] src base width offset[4:0] STORE

- Store needs to read two registers, **rs1** for base memory address, and **rs2** for data to be stored, as well immediate offset!
- Can't have both **rs2** and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, **no rd**!
- RISC-V design decision is to move low 5 bits of immediate to where **rd** field was in other instructions – keep **rs1/rs2** fields in same place
 - Register names more critical than immediate bits in hardware design

8/27/2021

37

S-Format Example

- RISC-V Assembly Instruction:

sw x14, 8(x2)

31 Imm[11:5]	25-24 rs2	20-19 rs1	15-14 funct3	12-11 imm[4:0]	7-6 opcode	0
-----------------	--------------	--------------	-----------------	-------------------	---------------	---

offset[11:5] src base width offset[4:0] STORE

0000000	01110	00010	010	01000	0100011	
---------	-------	-------	-----	-------	---------	--

offset[11:5] offset[4:0]
=0 rs2=14 rs1=2 SW =8 STORE

0000000 01000 combined 12-bit offset = 8

8/27/2021

38

All RV32 Store Instructions

- Store byte, halfword, word

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

width

8/27/2021

39

RISC-V Conditional Branches

- E.g., **beq x1, x2, Label**
- Branches read two registers but don't write to a register (similar to stores)
- How to encode label, i.e., where to branch to?

8/27/2021

40

Branching Instruction Usage

- Branches typically used for loops (**if-else**, **while**, **for**)
 - Loops are generally small (< 50 instructions)
 - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
 - Largest branch distance limited by size of code
 - Address of current instruction stored in the program counter (PC)

8/27/2021

41

PC-Relative Addressing

- **PC-Relative Addressing:** Use the **immediate** field as a two's complement offset to PC
 - Branches generally change the PC by a small amount
 - Can specify $\pm 2^{11}$ 'unit' addresses from the PC
 - (We will see in a bit that we can encode 12-bit offsets as immediates)
- Why not use byte as a unit of offset from PC?
 - Because instructions are 32-bits (4-bytes)
 - We don't branch into middle of instruction

8/27/2021

42

Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC
- This would allow one branch instruction to reach $\pm 2^{11} \times 32$ -bit instructions either side of PC
 - Four times greater reach than using byte offset

8/27/2021

43

Branch Calculation

- If we **don't** take the branch:
$$PC = PC + 4$$
 (i.e., next instruction)
- If we **do** take the branch:
$$PC = PC + immediate * 4$$
- Observations:
 - **immediate** is number of instructions to jump (remember, specifies words) either forward (+) or backwards (-)

8/27/2021

44

RISC-V Feature, n × 16-bit Instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that $\frac{1}{2}$ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32\text{-bit instructions}$ on either side of PC

8/27/2021

45

RISC-V B-Format for Branches

0011111	01011	01010	001	01000	1100111
imm[12:6]	rs2	rs1	funct3	imm[5:1]	opcode

- B-format is mostly same as S-Format, with two register sources (**rs1/rs2**) and a 12-bit immediate **imm[12:1]**
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

8/27/2021

46

Branch Example, Determine Offset

- RISC-V Code:

```
Loop: beq x19,x10,End
      add x18,x18,x10
      addi x19,x19,-1
      j Loop
End: # target instruction
```

Count instructions from branch

1
2
3
4

- Branch offset =

4 × 32-bit instructions = 16 bytes

- (Branch with offset of 0, branches to itself)

8/27/2021

47



Branch Example, Determine Offset

- RISC-V Code:

```
Loop: beq x19,x10,End
      add x18,x18,x10
      addi x19,x19,-1
      j Loop
End: # target instruction
```

Count instructions from branch

1
2
3
4

?????? 01010 10011 000 ???? 1100011
imm rs2=10 rs1=19 BEQ imm BRANCH

8/27/2021

48

Branch Example, Determine Offset

- RISC-V Code:

```

Loop: beq x19,x10,End
      add x18,x18,x10
      addi x19,x19,-1
      j Loop
End: # target instruction
  
```

Offset = 16 bytes
= 8 x 2

					01000
???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

8/27/2021

49

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]								0	R-type
31	30	25 24	20 19	15 14	12 11	8 7 6	rd	opcode	
funct7		rs2	rs1		funct3	rd		opcode	I-type
imm[11:0]			rs1	funct3		rd		opcode	S-type
imm[11:5]	rs2		rs1	funct3		imm[4:0]	opcode		B-type
imm[12 10:5]	rs2		rs1	funct3		imm[4:1 11]	opcode		

32-bit immediates produced, imm[31:0]

31	25 24	12 11	10	5	4	1	0	
-inst[31]-		inst[30:25]	inst[24:21]	inst[20]				I-imm.
-inst[31]-		inst[30:25]	inst[11:8]	inst[7]				S-imm.
-inst[31]-	inst[7]	inst[30:25]	inst[11:8]	0				B-imm.

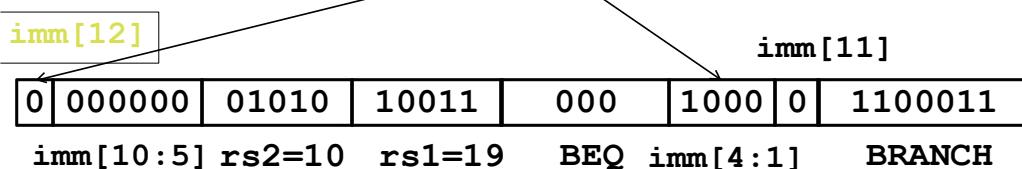
Upper bits sign-extended from inst[31] always Only bit 7 of instruction changes role in immediate between Sand B

8/27/2021 50

Branch Example, Complete Encoding

beq x19,x10, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16
imm[0] discarded, always zero



8/27/2021

51

All RISC-V Branch Instructions

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	beq
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	bne
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	blt
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	bge
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	bltu
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	bgeu

8/27/2021

52

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
 - If moving individual lines of code, then yes
 - If moving all of code, then no ('position-independent code')
- What do we do if destination is $> 2^{10}$ instructions away from branch?
 - Other instructions save us

8/27/2021

53

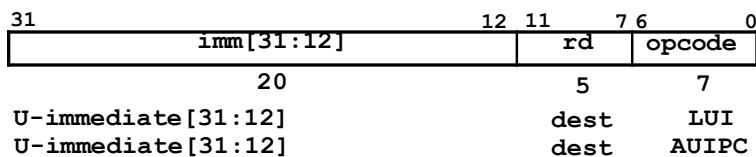
Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
 - If moving individual lines of code, then yes
 - If moving all of code, then no ('position-independent code')
- What do we do if destination is $> 2^{10}$ instructions away from branch?
 - Other instructions save us
 - `beq x10, x0, far`
 - # next instr → `bne x10, x0, next`
 - `j far`
 - `next: # next instr`

8/27/2021

54

U-Format for “Upper Immediate” Instructions



- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
 - **lui** – Load Upper Immediate
 - **auipc** – Add Upper Immediate to PC

8/27/2021

55

LUI to Create Long Immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an **addi** to set low 12 bits, can create any 32-bit value in a register using two instructions (**lui**/**addi**).

```
lui x10, 0x87654          # x10 = 0x87654000  
addi x10, x10, 0x321      # x10 = 0x87654321
```

56

One Corner Case

How to set **0xDEADBEEF**?

```
lui x10, 0xDEADB      # x10 = 0xDEADB000  
addi x10, x10, 0xEEF # x10 = 0xDEADAEEF
```

addi 12-bit immediate is always sign-extended, if top bit is set, will subtract -1 from upper 20 bits

8/27/2021

57

Solution

How to set **0xDEADBEEF**?

```
LUI x10, 0xDEADC      # x10 = 0xDEADC000  
  
ADDI x10, x10, 0xEEF # x10 =  
#0xDEADBEEF
```

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

```
li x10, 0xDEADBEEF # Creates two  
#instructions
```

8/27/2021

58

AUIPC

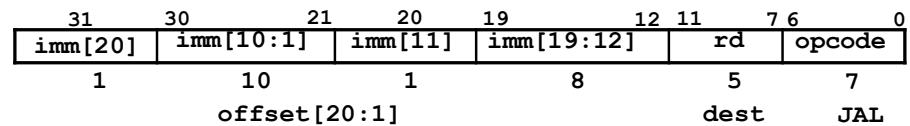
- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

```
Label: AUIPC x10, 0 # Puts address of  
# Label in x10
```

8/27/2021

59

J-Format for Jump Instructions



- **jal** saves PC+4 in register **rd** (the return address)
 - Assembler “j” jump is pseudo-instruction, uses JAL but sets **rd=x0** to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

8/27/2021

60

Uses of JAL

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return
address

# Call function within 218 instructions
of PC
jal ra, FuncName
```

JALR Instruction (I-Format)

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	func3	rd	opcode	
12	5	3	5	7	
offset[11:0]	base	0	dest	JALR	

- **jalr rd, rs, immediate**

- Writes PC+4 to rd (return address)
- Sets PC = **rs + immediate**
- Uses same immediates as arithmetic and loads
 - **no** multiplication by 2 bytes
 - In contrast to branches and **jal**

8/27/2021

61

8/27/2021

62

Uses of JALR

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit absolute
address
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```

Summary of RISC-V Instruction Formats

31	30	25 24	21	20 19	15 14	12 11	8 7 6	0
funct7	rs2		rs1	funct3		rd	opcode	R-type
		imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2		rs1	funct3	imm[4:0]	opcode		S-type
imm[12:10:5]	rs2		rs1	funct3	im	m[4:1 11]	opcode	B-type
		imm[31:12]			rd	opcode		U-type
imm[20:10:1 11]		imm[19:12]		rd	opcode			J-type

8/27/2021

63

8/27/2021

64

Complete RV32I ISA

Open Reference Card™					
Base Integer Instructions-RV32I					
Category	Name	Fmt	RV32I Base		
Shifts	Shift Left Logical	R	SLL rd,r1,n2		
	Shift Left Log. Imm.	I	SLLI rd,r1,n2,shamt		
	Shift Right Logical	R	SRL rd,r1,n2		
	Shift Right Log. Imm.	I	SRRL rd,r1,n2,shamt		
	Shift Right Arithmetic	R	SRA rd,r1,n2		
	Shift Right Arith. Imm.	I	SRAI rd,r1,n2,shamt		
Arithmetic ADD	Add	R	ADD rd,r1,n2		
	ADD Immediate	I	ADDI rd,r1,imm		
	Subtract	R	SUB rd,r1,n2		
	Load Upper Imm	U	LUU rd,imm		
	Add Upper Imm to PC	U	AUIPC rd,imm		
Logical	XOR	R	XOR rd,r1,n2	Loads	Load Byte
	XOR Immediate	I	XORI rd,r1,imm		Load Halfword
	OR	R	OR rd,r1,n2		Load Byte Unsigned
	OR Immediate	I	ORI rd,r1,imm		Load Half Unsigned
	AND	R	AND rd,r1,n2		Load Word
	AND Immediate	I	ANDI rd,r1,imm	Stores	Store Byte
Comparisons	Set <	R	SLT rd,r1,r2		Store Halfword
	Set < Immediate	I	SLTI rd,r1,imm		Store Word
	Set < Unsigned	R	SLTU rd,r1,r2		
	Set < Immediate Unsigned	I	SLTUI rd,r1,imm		
Branches	Branch =	B	BED rs1,r2,imm	Synch	Synch thread
	Branch ≠	B	BNE rs1,r2,imm		
	Branch <	B	BLT rs1,r2,imm	Environment CALL	I ECALL
	Branch ≥	B	BLTU rs1,r2,imm	BREAK	I EBREAK
Jump & Link	Jump	J	JAL rd,imm		
	Jump & Link Register	I	JALR rd,r1,imm		

8/27/2021

65

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great Ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolic, © 2020, UC Berkeley]

8/27/2021

1

RISC-V Processor Design

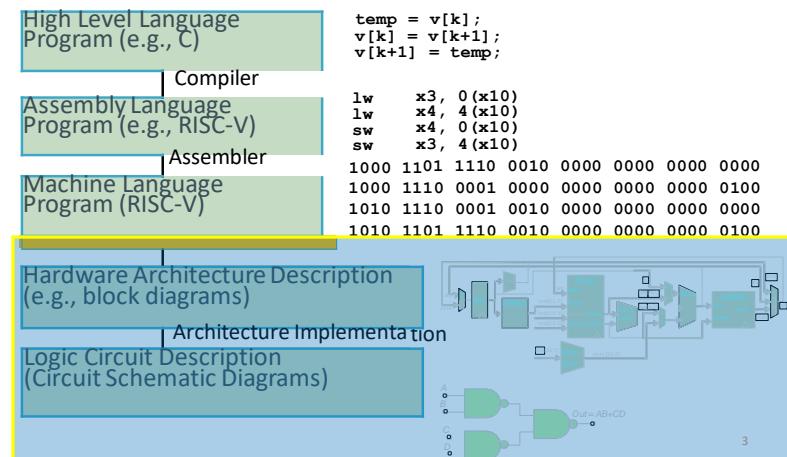
8/27/2021

2

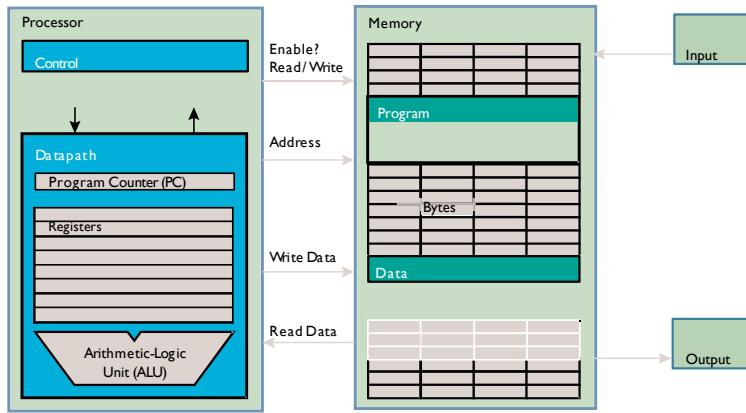
8/27/2021

3

Great Idea #1: Abstraction (Levels of Representation/Interpretation)



Our Single-Core Processor So Far...



8/27/2021

4

The CPU

- **Processor (CPU):** the active part of the computer that does all the work (data manipulation and decision-making)
- **Datapath:** portion of the processor that contains hardware necessary to perform operations required by the processor (the brawn)
- **Control:** portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)

8/27/2021

5

Need to Implement All RV32I Instructions

Open Reference Card					
Base Integer Instructions: RV32I					
Category Name	Fmt	RV32I Base	Category	Name	Fmt
Shifts Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I
Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I
Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I
Shift Right Log. Imm.	I	SR LI rd,rs1,shamt		Load Half Unsigned	I
Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I
Shift right. Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S
Arithmetic ADD	R	ADD rd,rs1,rs2		Store Halfword	S
ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S
Subtract	R	SUB rd,rs1,rs2	Branches	Branch =	B
	R	SUB rd,rs1,rs2		BEQ rs1,rs2,imm	B
	R	SUB rd,rs1,rs2		BNE rs1,rs2,imm	B
	R	SUB rd,rs1,rs2		BLT rs1,rs2,imm	B
	R	SUB rd,rs1,rs2		BGE rs1,rs2,imm	B
	R	SUB rd,rs1,rs2		BLTU rs1,rs2,imm	B
	R	SUB rd,rs1,rs2		BGEU rs1,rs2,imm	B
Logical Load Upper Imm to PC	U	LUI rd,imm	Jump & Link	J	JAL rd,imm
Add Upper Imm to PC	U	AUIPC rd,imm		JALR rd,rs1,imm	I
XOR	R	XOR rd,rs1,rs2			
XOR Immediate	I	XORI rd,rs1,imm			
OR	R	OR rd,rs1,rs2			
OR Immediate	I	ORI rd,rs1,imm			
AND	R	AND rd,rs1,rs2			
AND Immediate	I	ANDI rd,rs1,imm			
Compare Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I
Set < Immediate	I	SLTI rd,rs1,imm			
Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL BREAK	I
Set < Imm Unsigned	I	SLTIU rd,rs1,imm			

8/27/2021

6

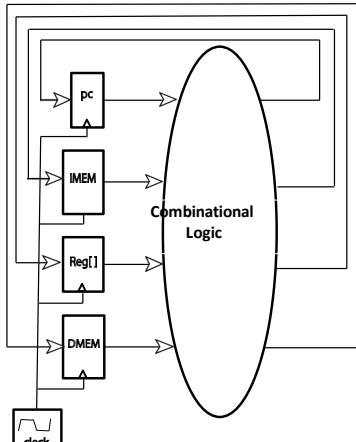


Building a RISC-V Processor

8/27/2021

7

One-Instruction-Per-Cycle RISC-V Machine



8/27/2021

8

- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settle at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

Stages of the Datapath : Overview

- Problem: a single, “monolithic” block that “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- Solution: break up the process of “executing an instruction” into stages, and then connect the stages to create the whole datapath
 - smaller stages are easier to design
 - easy to optimize (change) one stage without touching the others (modularity)

8/27/2021

9

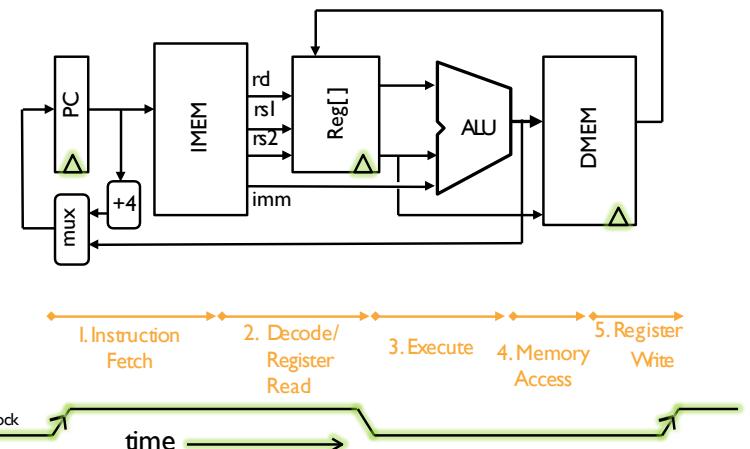
Five Stages of the Datapath

- Stage 1: Instruction Fetch (IF)**
- Stage 2: Instruction Decode (ID)**
- Stage 3: Execute (EX) - ALU (Arithmetic-Logic Unit)**
- Stage 4: Memory Access (MEM)**
- Stage 5: Write Back to Register (WB)**

8/27/2021

10

Basic Phases of Instruction Execution

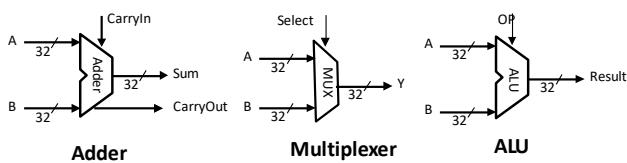


8/27/2021

11

Datapath Components: Combinational

- Combinational elements



- Storage elements + clocking methodology
- Building blocks

8/27/2021

12

8/27/2021

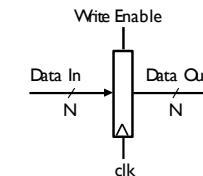
13

Datapath Elements: State and Sequencing (1/3)

- Register

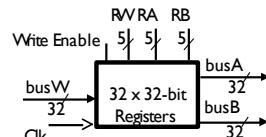
- Write Enable:

- Low (or deasserted) (0): Data Out will not change
- Asserted (1): Data Out will become Data In on positive edge of clock



Datapath Elements: State and Sequencing (2/3)

- Register file (regfile, RF) consists of 32 registers:
 - Two 32-bit output busses: busA and busB
 - One 32-bit input bus: busW
- Register is selected by:
 - RA (number) selects the register to put on busA (data)
 - RB (number) selects the register to put on busB (data)
 - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (Clk)
 - Clk input is a factor ONLY during write operation
 - During read operation, behaves as a combinational logic block:
 - RA or RB valid \Rightarrow busA or busB valid after “access time.”



8/27/2021

14

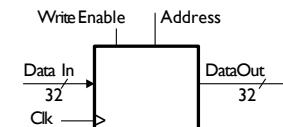
8/27/2021

15

Datapath Elements: State and Sequencing (3/3)

- “Magic” Memory

- One input bus: Data In
- One output bus: Data Out



- Memory word is found by:

- For Read: Address selects the word to put on Data Out
- For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus

- Clock input (CLK)

- CLK input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block: Address valid \Rightarrow Data Out valid after “access time”

State Required by RV32I ISA (1/2)

Each instruction during execution reads and updates the state of :

(1) Registers, (2) Program counter, (3) Memory

- Registers (**x0 . . x31**)
 - Register file (*regfile*) **Reg** holds 32 registers x 32 bits/register:
Reg[0] . . Reg[31]
 - First register read specified by **rs1** field in instruction
 - Second register read specified by **rs2** field in instruction
 - Write register (destination) specified by **rd** field in instruction
 - **x0** is always 0 (writes to **Reg[0]** are ignored)
- Program Counter (**PC**)
 - Holds address of current instruction

State Required by RV32I ISA (2/2)

■ Memory (**MEM**)

- Holds both instructions & data, in one 32-bit byte-addressed memory space
- We'll use separate memories for instructions (**IMEM**) and data (**DMEM**)
 - These are placeholders for instruction and data caches
- Instructions are read (*fetched*) from instruction memory (assume **IMEM** read-only)
- Load/store instructions access data memory

R-Type Add Datapath

Review: R-Type Instructions

R-format : ALU						
[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	
func7	rs2	rs1	func3	rd		opcode
0000000	rs2	rs1	000 : ADD	rd		0110011:OP-R
0100000	rs2	rs1	000 : SUB	rd		0110011:OP-R
0000000	rs2	rs1	001 : SLL	rd		0110011:OP-R
0000000	rs2	rs1	010 : SLT	rd		0110011:OP-R
0000000	rs2	rs1	011 : SLTU	rd		0110011:OP-R
0000000	rs2	rs1	100 : XOR	rd		0110011:OP-R
0000000	rs2	rs1	101 : SRL	rd		0110011:OP-R
0100000	rs2	rs1	101 : SRA	rd		0110011:OP-R
0000000	rs2	rs1	110 : OR	rd		0110011:OP-R
0000000	rs2	rs1	111 : AND	rd		0110011:OP-R

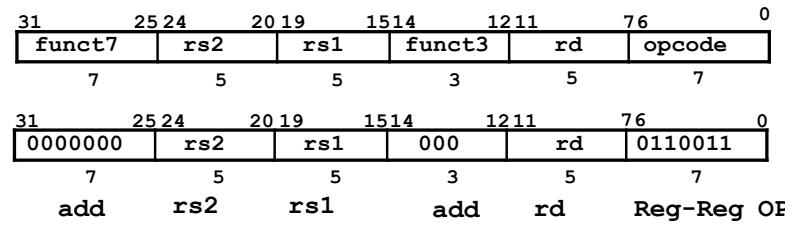
- E.g. Addition/subtraction **add rd, rs1, rs2**

$$R[rd] = R[rs1] + R[rs2]$$

$$\text{sub } rd, rs1, rs2$$

$$R[rd] = R[rs1] - R[rs2]$$

Implementing the add instruction



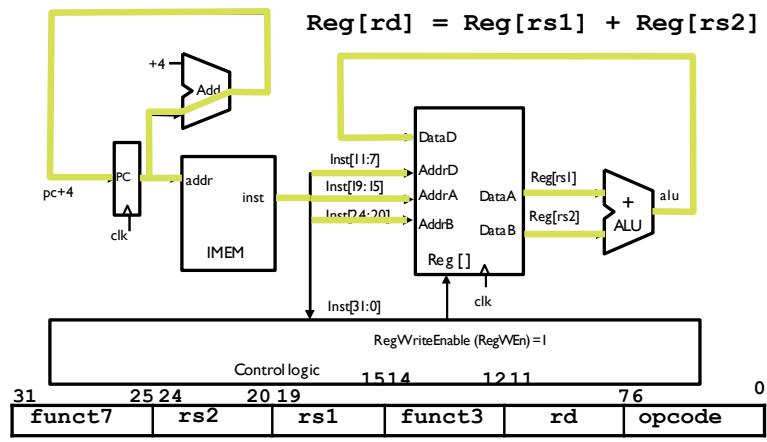
add rd, rs1, rs2

- Instruction makes two changes to machine's state:
 - $\text{Reg}[rd] = \text{Reg}[rs1] + \text{Reg}[rs2]$
 - $\text{PC} = \text{PC} + 4$

8/27/2021

20

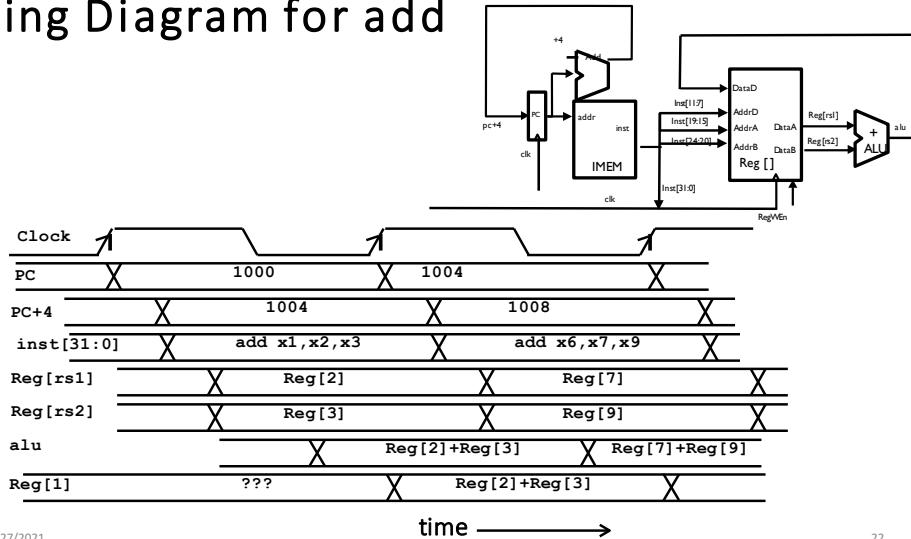
Datapath for add



8/27/2021

21

Timing Diagram for add



8/27/2021

22

Sub Datapath

8/27/2021

23

Implementing the sub instruction

0000000	rs2	rs1	000	rd	0110011
0100000	rs2	rs1	000	rd	0110011

add
sub

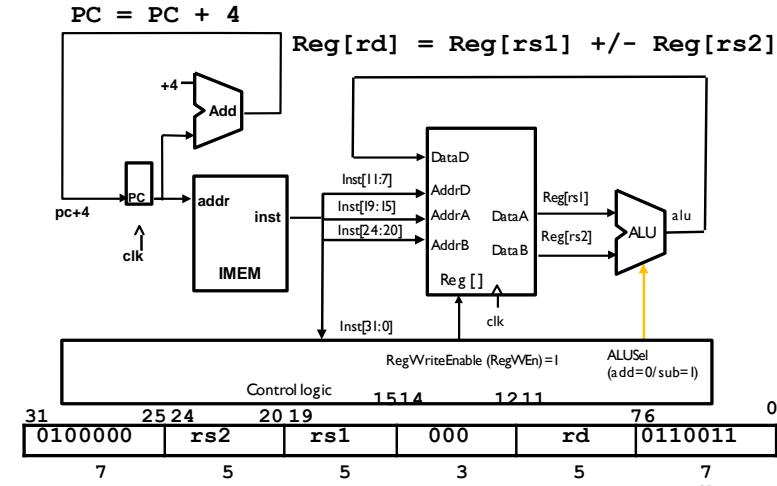
sub rd, rs1, rs2

- Almost the same as add, except now have to subtract operands instead of adding them
- inst[30]** selects between add and subtract

8/27/2021

24

Datapath for add/sub



8/27/2021

24

Implementing Other R-Format Instructions

0000000	rs2	rs1	000	rd	0110011
0100000	rs2	rs1	000	rd	0110011
0000000	rs2	rs1	001	rd	0110011
0000000	rs2	rs1	010	rd	0110011
0000000	rs2	rs1	011	rd	0110011
0000000	rs2	rs1	100	rd	0110011
0000000	rs2	rs1	101	rd	0110011
0100000	rs2	rs1	101	rd	0110011
0000000	rs2	rs1	110	rd	0110011
0000000	rs2	rs1	111	rd	0110011

add
sub
sll
slt
sltu
xor
srl
sra
or
and

All implemented by decoding funct3 and funct7 fields
and selecting appropriate ALU function

8/27/2021

26

8/27/2021

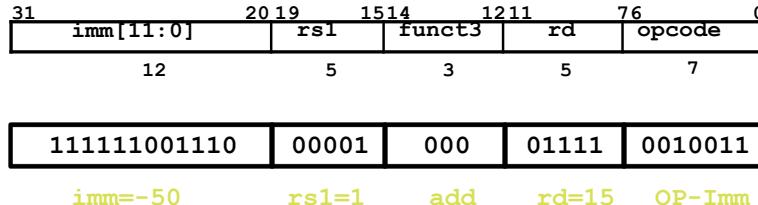
27

Datapath With Immediates

Implementing I-Format - addi instruction

- RISC-V Assembly Instruction:

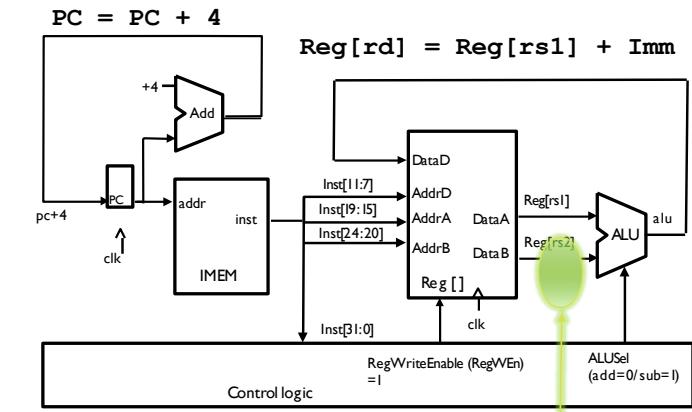
addi x15,x1,-50



8/27/2021

28

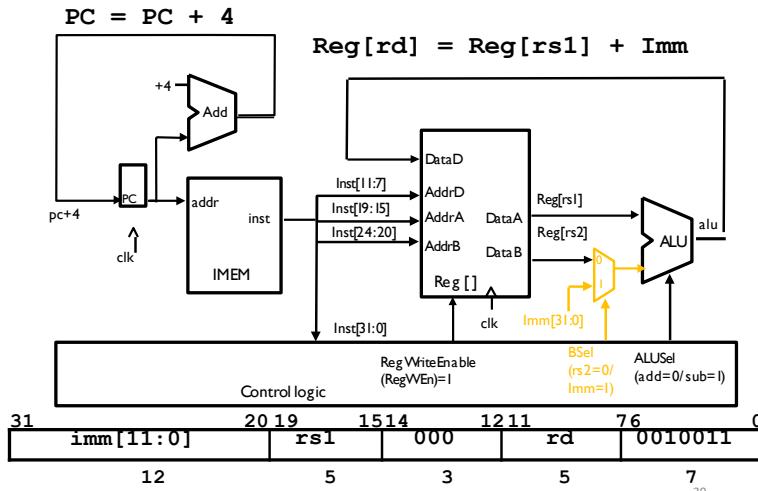
Datapath for add/sub



8/27/2021

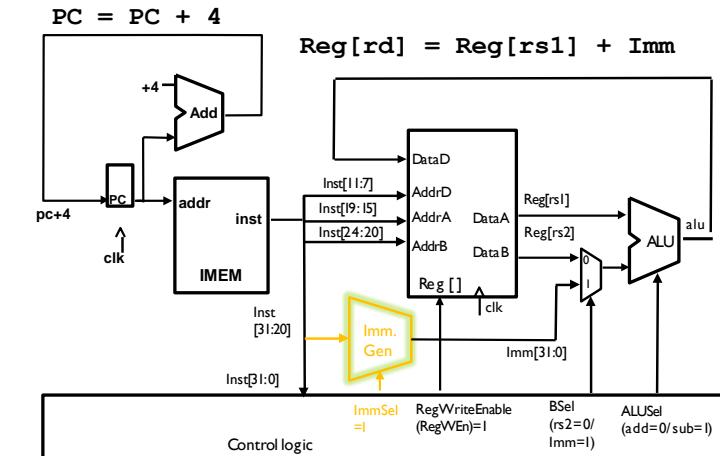
29

Adding **addi** to Datapath



8/27/2021

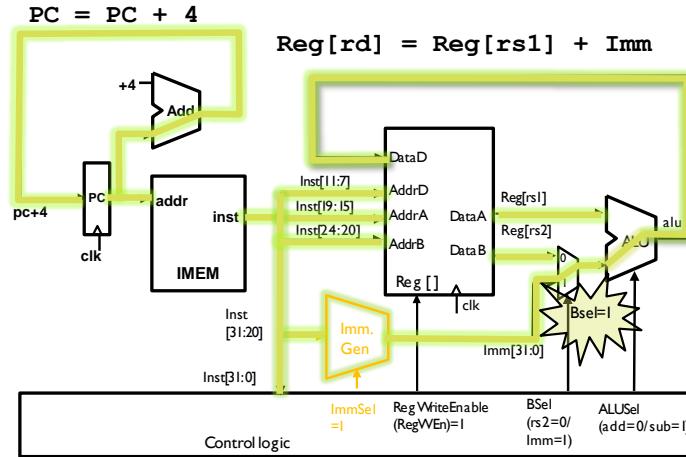
Adding **addi** to Datapath



8/27/2021

31

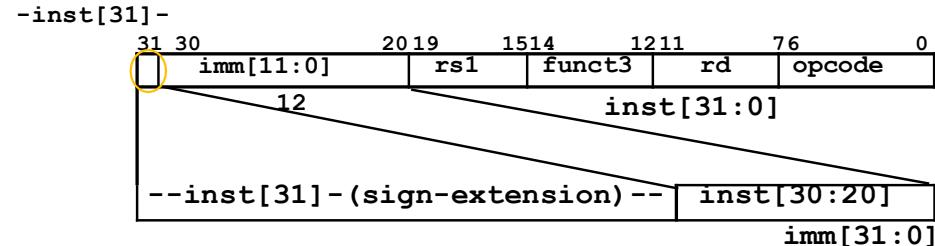
Adding **addi** to Datapath



8/27/2021

32

I-Format Immediates

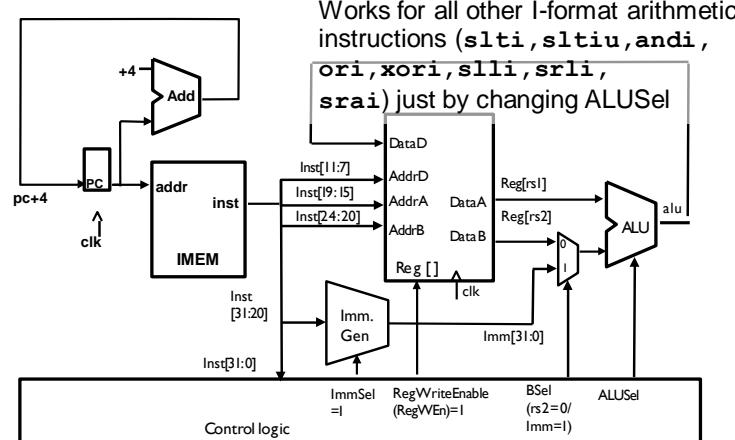


- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

8/27/2021

33

Adding **addi** to Datapath



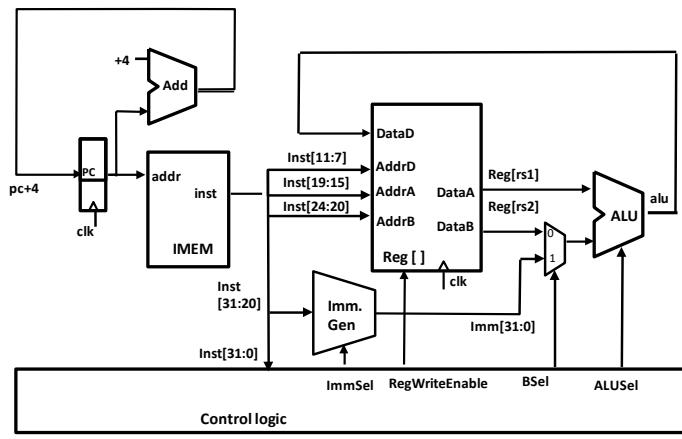
8/27/2021

34

Supporting Loads

35

R+I Arithmetic/Logic Datapath



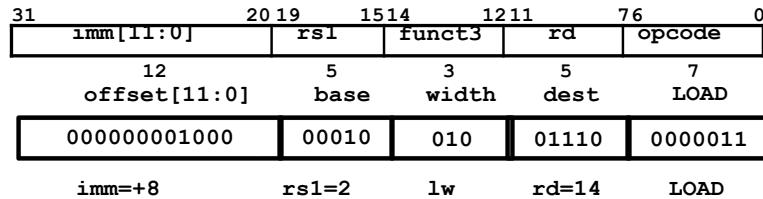
8/27/2021

RISC-V (37)

36

Add **lw**

- RISC-V Assembly Instruction (I-type): **lw x14, 8(x2)**



- The 12-bit signed immediate is added to the base address in register **rs1** to form the memory address
 - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register **rd**

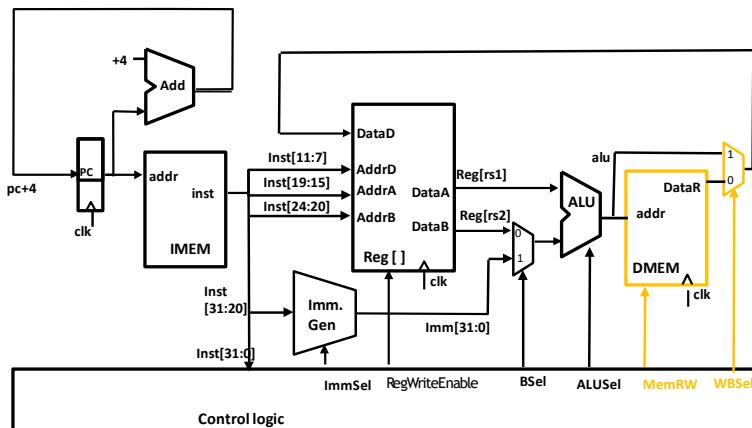
8/27/2021

RISC-V

(38)

37

R+I Arithmetic/Logic Datapath

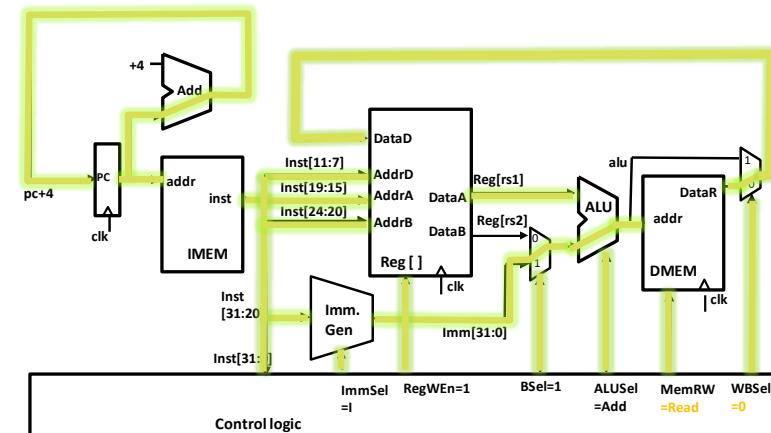


8/27/2021

RISC-V (39)

38

R+I Arithmetic/Logic Datapath



8/27/2021

RISC-V (40)

39

All RV32 Load Instructions

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

- funct3 field encodes size and ‘signedness’ of load data

- Supporting the narrower loads requires additional logic to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
 - It is just a mux + a few gates

8/27/2021

RISC-V (41)

40

8/27/2021

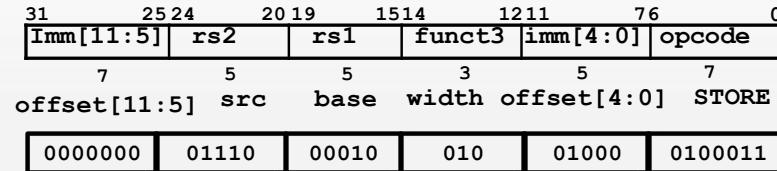
41

Datapath for Stores

Adding **sw** instruction

sw: Reads two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!

sw x14, 8(x2)



offset[11:5] = 0 rs2=14 rs1=2 SW = 8 STORE

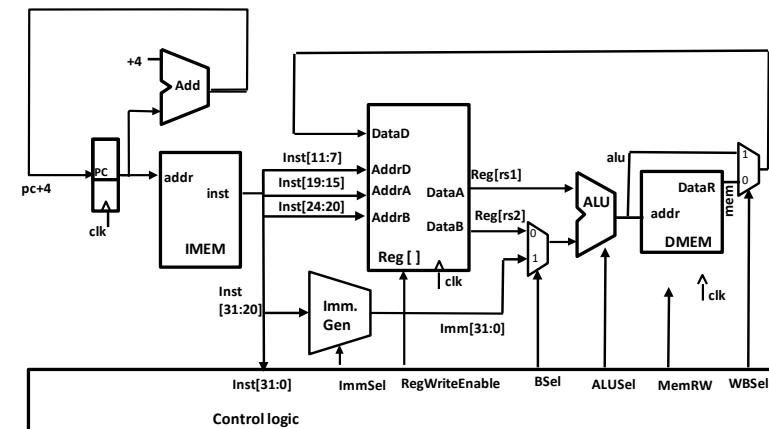
0000000 01000 combined 12-bit offset = 8

8/27/2021

RISC-V (43)

42

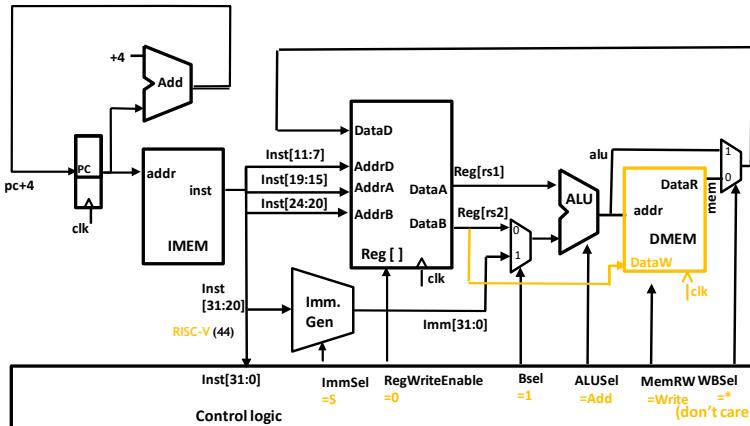
Datapath with **lw**



RISC-V (44)

43

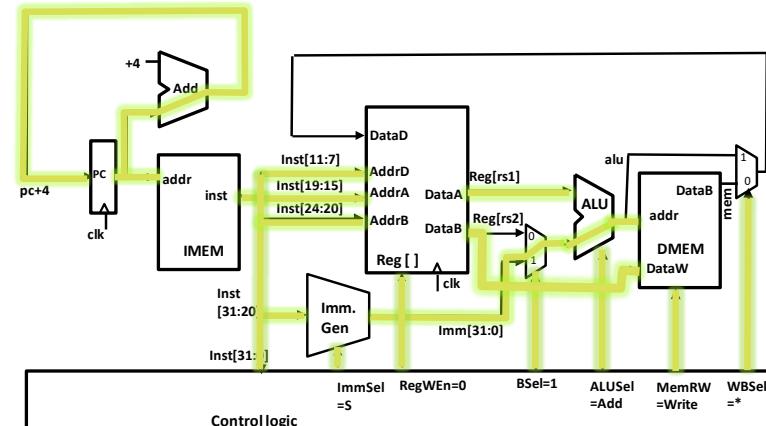
Adding sw to Datapath



8/27/2021

RISC-V (44)

Adding sw to Datapath

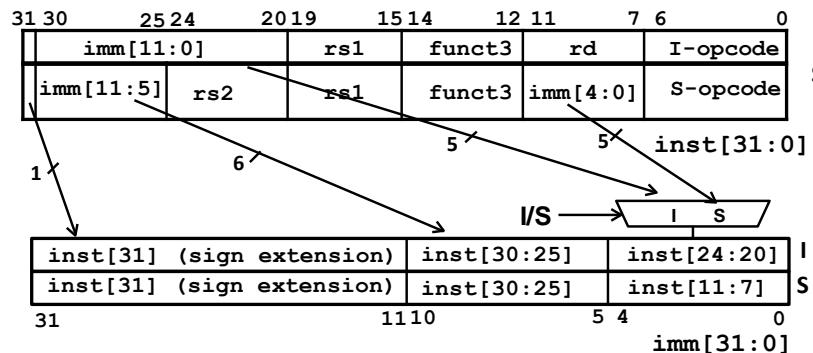


8/27/2021

RISC-V (46)

45

I+S Immediate Generation



- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction

8/27/2021

46

8/27/2021

Berkeley

RISC-V (47)

47

All RV32 Store Instructions

- Store byte writes the low byte to memory
- Store halfword writes the lower two bytes to memory

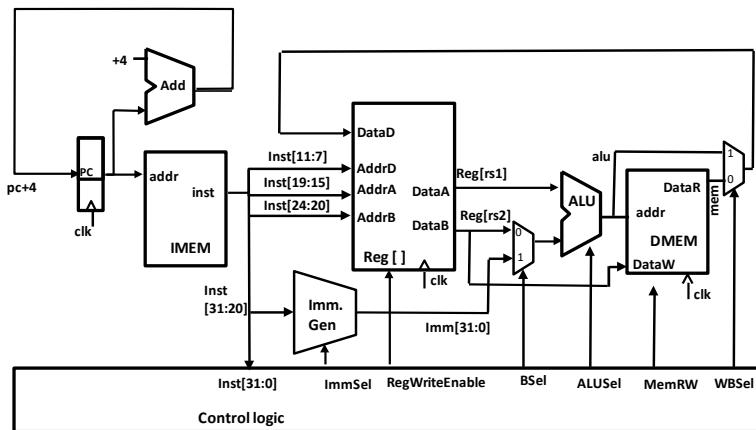
Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw
width						

RISC-V B-Format for Branches

31	30	25	24	20	19	15	14	12	11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode						
1	6	5	5	3	4	1	7						
	offset[12 11:5]		rs1	funct3				BRANCH					
			rs2		offset[4:1 11]								

- B-format is mostly same as S-Format, with two register sources (**rs1/rs2**) and a 12-bit immediate **imm[12:1]**
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode **even** 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

Datapath So Far

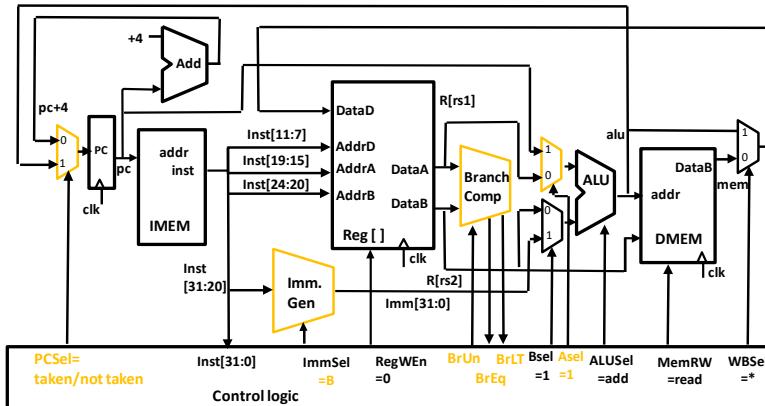


To Add Branches

- Different change to the state:

$$\begin{cases} \text{PC} + 4, & \text{branch not taken} \\ \text{PC} + \text{immediate}, & \text{branch taken} \end{cases}$$
- Six branch instructions: **beq**, **bne**, **blt**, **bge**, **bltu**, **bgeu**
- Need to compute **PC + immediate** and to compare values of **rs1** and **rs2**
 - But have only one ALU – need more hardware

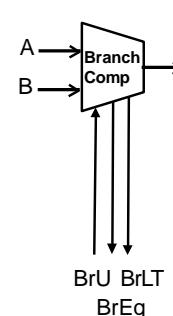
Adding Branches



8/27/2021

RISC-V (52)

Branch Comparator



- BrEq** = 1, if $A=B$
- BrLT** = 1, if $A < B$
- BrUn** = 1 selects unsigned comparison for **BrLT**, 0=signed

BGE branch: $A \geq B$, if $A < B$

$$A < B = !(A < B)$$

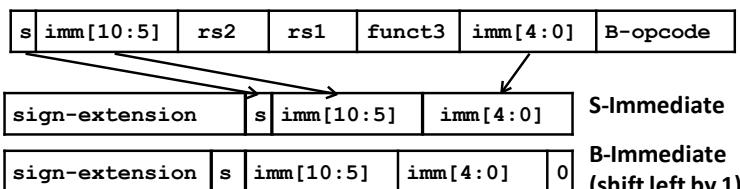
8/27/2021

Berkeley

RISC-V (53)

Branch Immediates (In Other ISAs)

- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes
- Standard approach: Treat immediate as in range -2048..+2047, then shift left by 1 bit to multiply by 2 for branches



Each instruction immediate bit can appear in one of two places in output immediate value – so need one 2-way mux per bit

8/27/2021

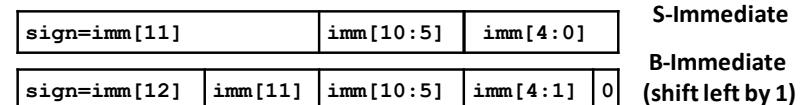
Berkeley

Branch Immediates (In RISC-V ISAs)

- 12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes

NOTE : Page. 116 (RISC V Textbook -> get extra information)

- RISC-V approach: keep 11 immediate bits in fixed position in output value, and rotate LSB of S-format to be bit 12 of B-format



Only one bit changes position between S and B, so only need a single-bit 2-way mux

The RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of halfwords between the branch and the branch target

8/27/2021

Berkeley

RISC-V Immediate Encoding

Instruction encodings, $\text{inst}[31:0]$

31	30	25 24	20 19	15 14	12 11	8 7 6	0
imm[11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12:10:5]		rs2	rs1	funct3	imm[4:1 11]	opcode	B-type

32-bit immediates produced, $\text{imm}[31:0]$

31	25 24	12 11	10	5 4	1	0
-inst[31]-		inst[30:25]	inst[24:21]	inst[20]		I-imm.
-inst[31]-		inst[30:25]	inst[11:8]	inst[7]		S-imm.
-inst[31]-	inst[7]	inst[30:25]	inst[11:8]	0		B-imm.

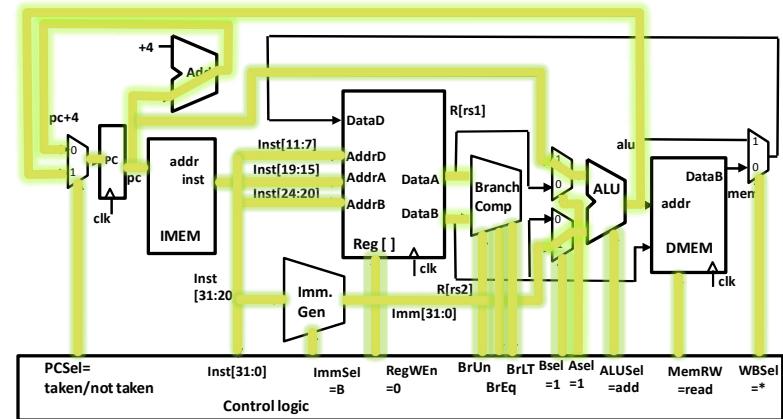
Upper bits sign-extended from $\text{inst}[31]$ always

Only bit 7 of instruction changes role in immediate between Sand B

8/27/2021

56

Lighting Up Branch Path



8/27/2021

57

Adding JALR to Datapath

8/27/2021

58

Let's Add JALR(I-Format)

31	20 19	15 14	func3	12 11	7 6	0
imm[11:0]		rs1		rd		opcode
12	5	3		0	5	7
offset[11:0]	base			dest	JALR	

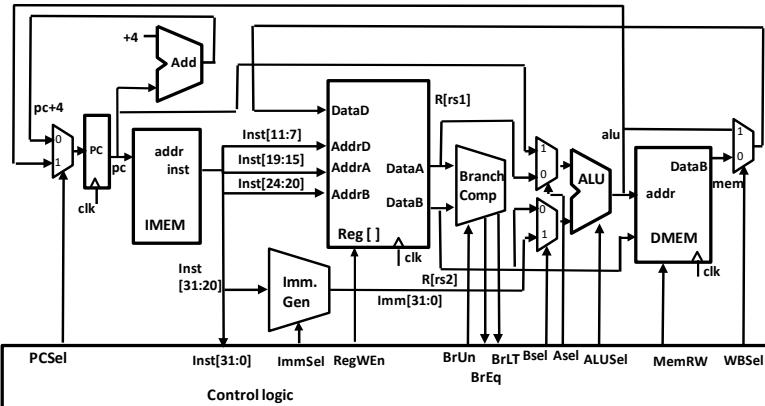
- **JALR rd, rs, immediate**
- Two changes to the state
 - Writes PC+4 to rd (return address)
 - Sets PC = rs1 + immediate
 - Uses same immediates as arithmetic and loads
 - **no** multiplication by 2 bytes
 - LSB is ignored

8/27/2021

RISC-V (60)

59

Datapath So Far, With Branches

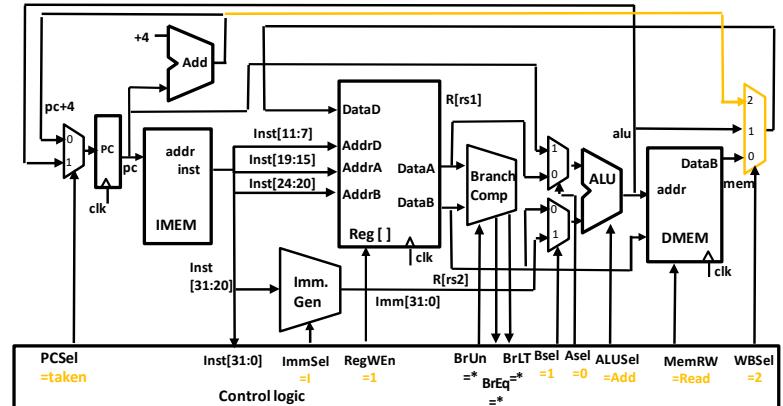


8/27/2021

RISC-V (61)

60

Datapath With JALR

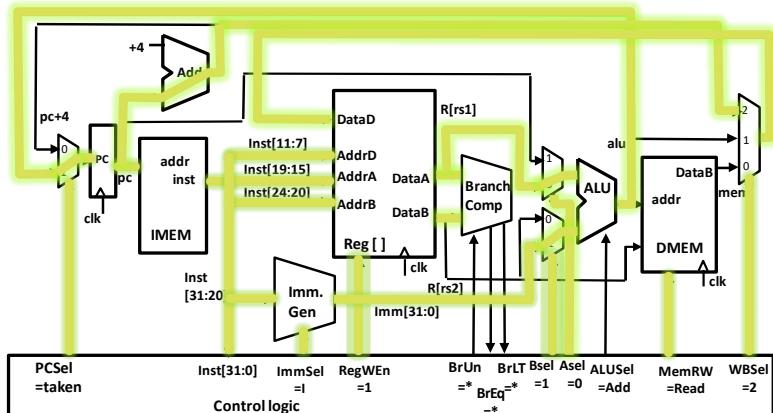


8/27/2021

RISC-V (62)

61

Datapath With JALR



8/27/2021

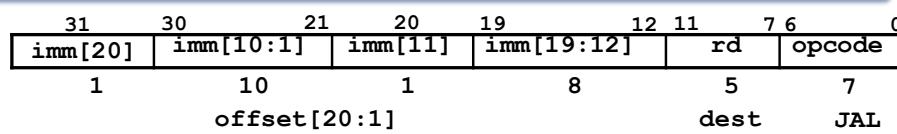
RISC-V (63)

62

Adding JAL

63

JFormat for Jump Instructions



- Two changes to the state
 - `jal` saves PC+4 in register `rd` (the return address)
 - Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

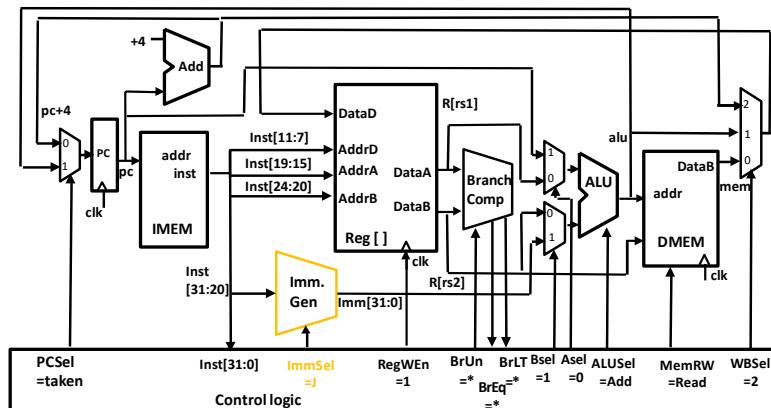
8/27/2021



RISC-V (65)

64

Datapath with JAL

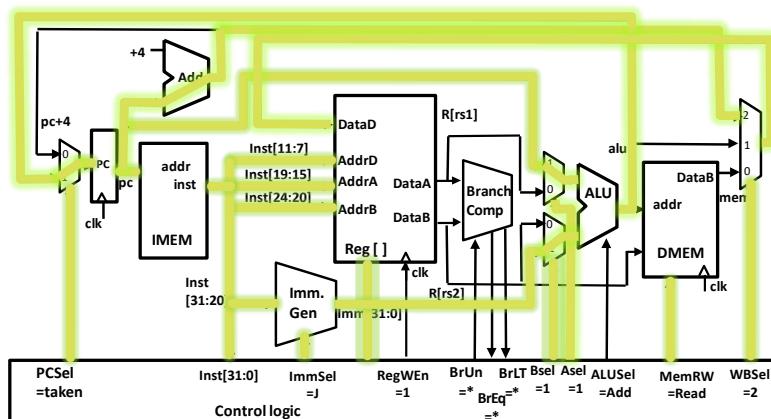


8/27/2021

RISC-V (66)

65

LightUp JAL Path



8/27/2021

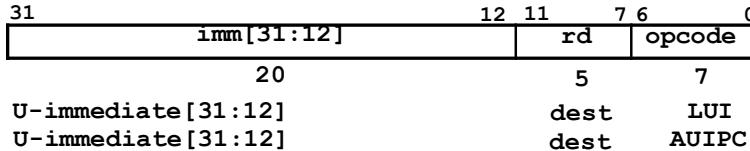
RISC-V (67)

66

**Adding
U-Types**

67

U-Format for “Upper Immediate” Instructions



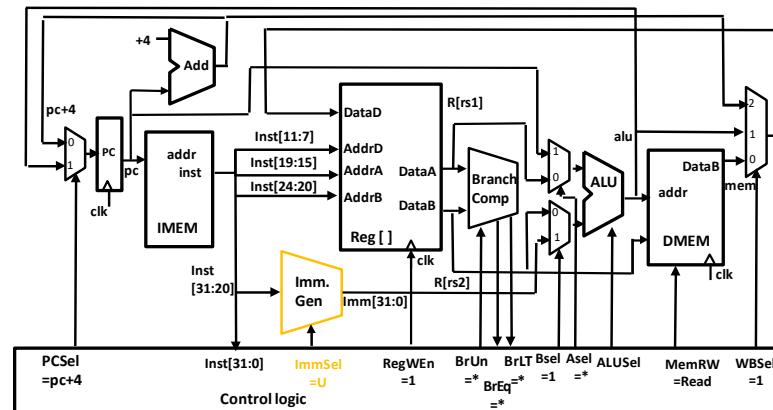
- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, **rd**
- Used for two instructions
 - lui** – Load Upper Immediate
 - auipc** – Add Upper Immediate to PC

8/27/2021

RISC-V (69)

68

Datapath With LUI, AUIPC

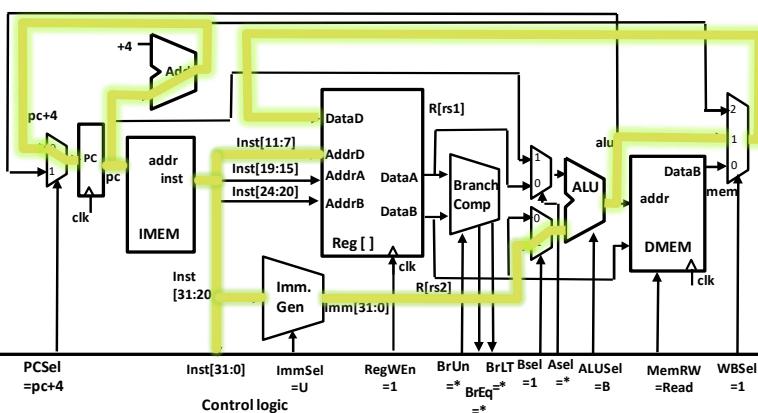


8/27/2021

RISC-V (70)

69

Lighting Up LUI

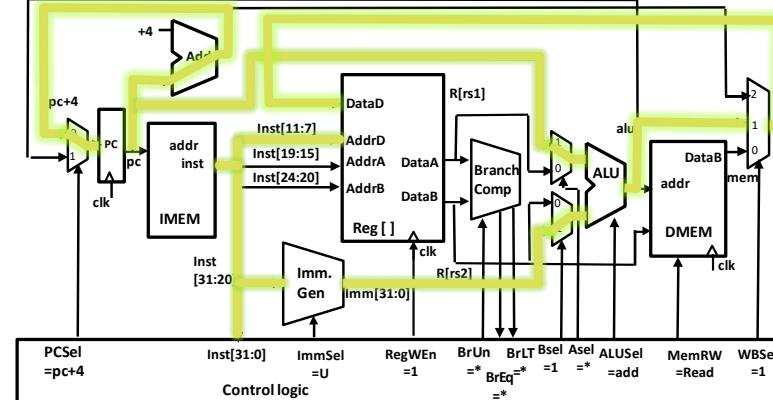


8/27/2021

RISC-V (71)

70

Lighting Up AUIPC

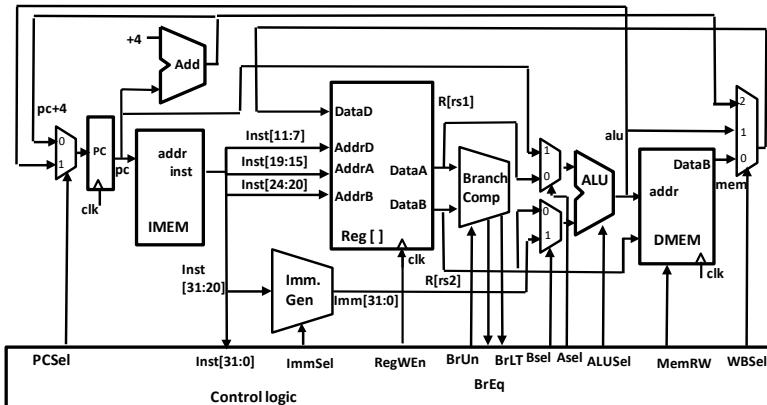


8/27/2021

RISC-V (72)

71

Complete RV32I Datapath!



8/27/2021

72

8/27/2021

73

RISC-V (74)

“And In Conclusion...”

Complete RV32I ISA!

OpenRISC Reference Card						
Base Integer Instructions: RV32I						
CategoryName	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts Shift Left Logical	R	SLL rd,rs1,rs2	Loads	Load Byte	I	LB rd,rs1,imm
Shift Left Log. Imm.	I	SLLI rd,rs1,shamt		Load Halfword	I	LH rd,rs1,imm
Shift Right Logical	R	SRL rd,rs1,rs2		Load Byte Unsigned	I	LBU rd,rs1,imm
Shift Right Log. Imm.	I	SRRI rd,rs1,shamt		Load Half Unsigned	I	LHU rd,rs1,imm
Shift Right Arithmetic	R	SRA rd,rs1,rs2		Load Word	I	LW rd,rs1,imm
Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Stores	Store Byte	S	SB rs1,rs2,imm
Arithmetic ADD	R	ADD rd,rs1,rs2		Store Halfword	S	SH rs1,rs2,imm
ADD Immediate	I	ADDI rd,rs1,imm		Store Word	S	SW rs1,rs2,imm
Subtract	R	SUB rd,rs1,rs2	Branches	Branch =	B	BEQ rs1,rs2,imm
Add Upper Imm to PC	U	LUI rd,imm		Branch ≠	B	BNE rs1,rs2,imm
Logical Load Upper Imm to PC	U	AUIPC rd,imm		Branch <	B	BLT rs1,rs2,imm
XOR Immediate	R	XOR rd,rs1,rs2		Branch ≥	B	BGE rs1,rs2,imm
XOR Immediate	I	XORI rd,rs1,imm		Branch < Unsigned	B	BLTU rs1,rs2,imm
OR Immediate	R	OR rd,rs1,rs2		Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
OR Immediate	I	ORI rd,rs1,imm	Jump & Link	JAL	J	JAL rd,imm
AND Immediate	R	AND rd,rs1,rs2		Jump & Link Register	I	JALR rd,rs1,imm
AND Immediate	I	ANDI rd,rs1,imm				
Compare Set <	R	SLT rd,rs1,rs2	Synch	Synch thread	I	FENCE
Set < Immediate	I	SLTI rd,rs1,imm				Not in 61C
Set < Unsigned	R	SLTU rd,rs1,rs2	Environment	CALL	I	ECALL
Set < Imm Unsigned	I	SLTIU rd,rs1,imm		BREAK	I	EBREAK

8/27/2021

RISC-V (75)



8/27/2021



RISC-V (76)

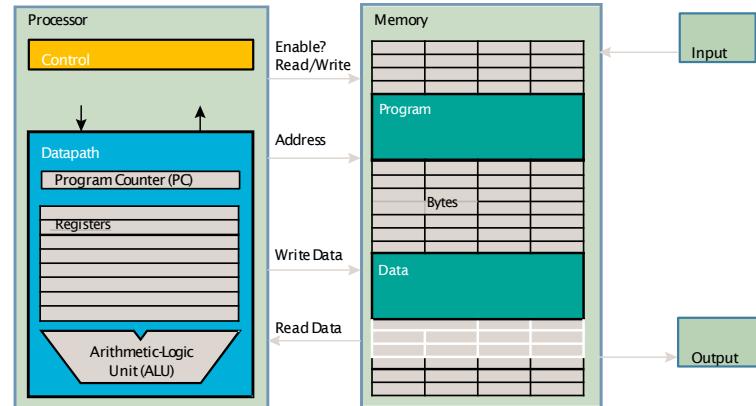
Review

- We have designed a complete datapath
 - Capable of executing all RISC-V instructions in one cycle each
 - Not all units (hardware) used by all instructions
- 5 Phases of execution
 - IF, ID, EX, MEM, WB
 - Not all instructions are active in all phases
- Controller specifies how to execute instructions
 - We still need to design it

75

Datapath Control

Our Single-Core Processor



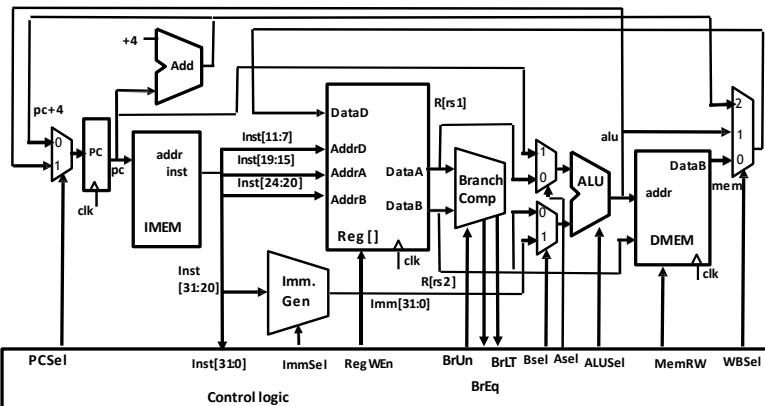
8/27/2021

83

8/27/2021

84

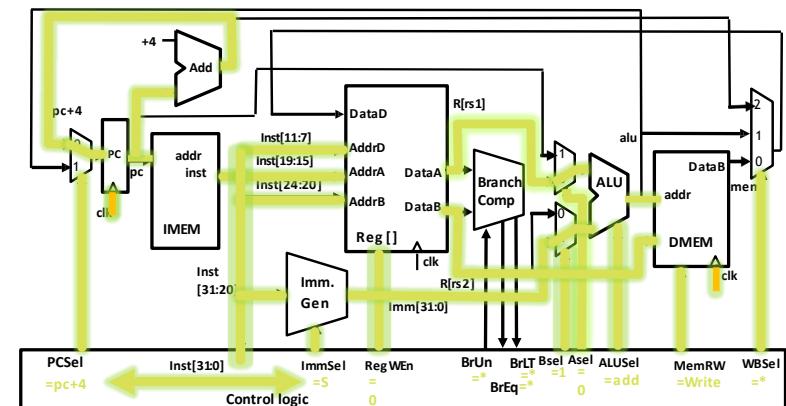
Single-Cycle RV32I Datapath and Control



8/27/2021

85

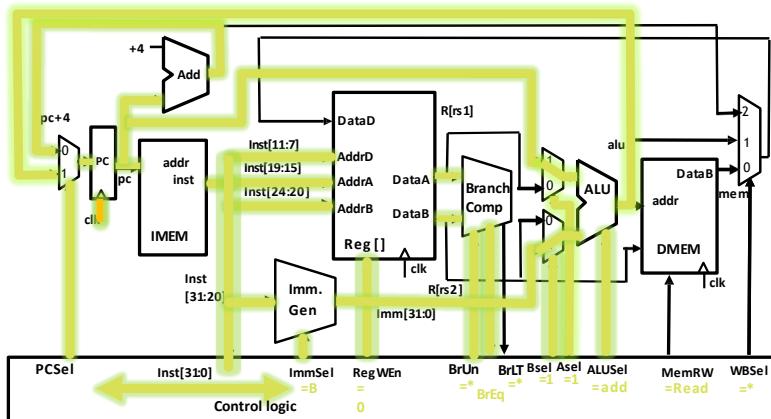
Example: **sw**



8/27/2021

86

Example: **beq**



8/27/2021

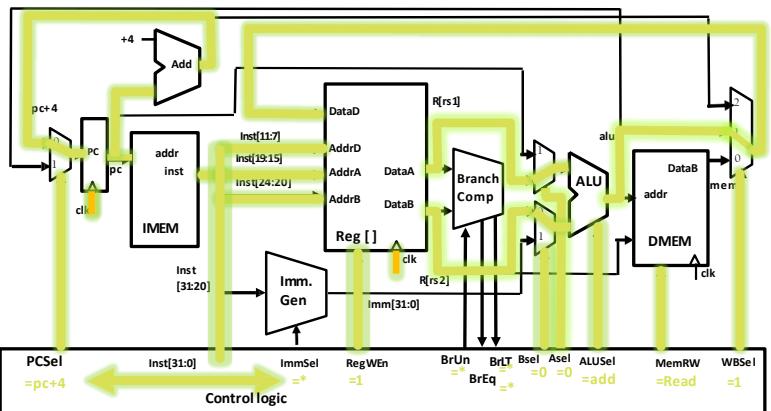
87

8/27/2021

88

Instruction Timing

Example: **add**

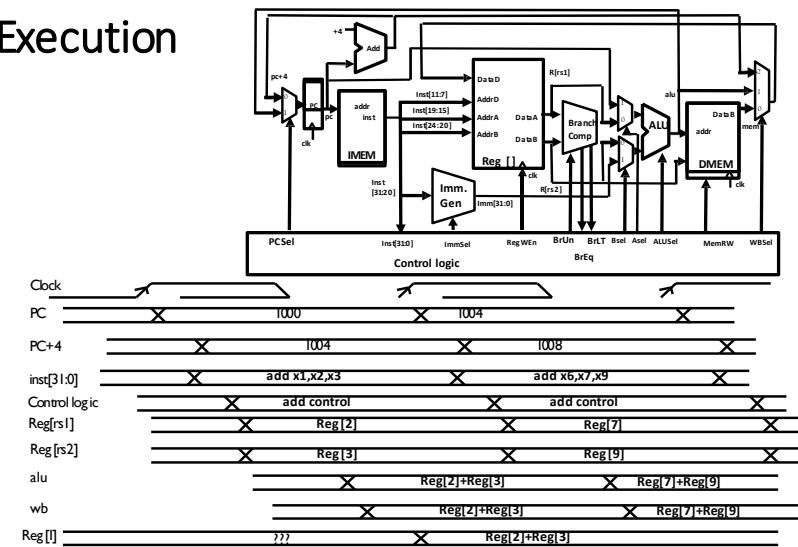


8/27/2021

89

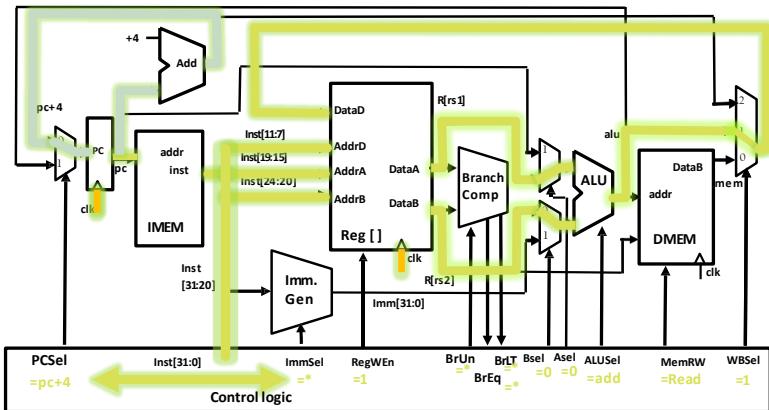
8/27/2021

Add Execution



90

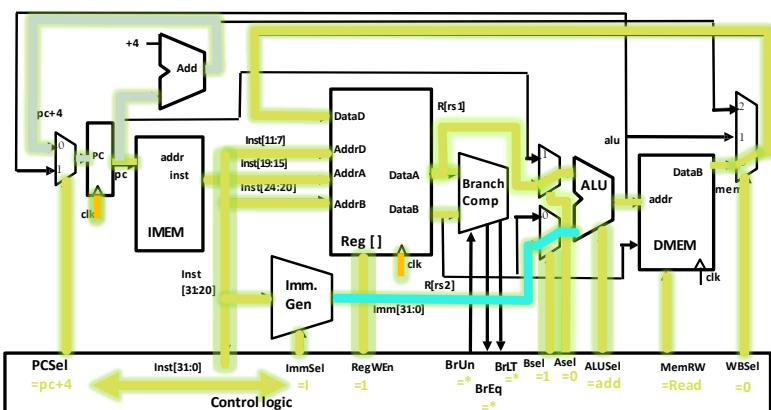
Example: add timing



$$\text{Critical path} = t_{\text{clk-q}} + \max \{ t_{\text{Add}} + t_{\text{mux}}, t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} \} + t_{\text{setup}}$$

$$= t_{\text{clk-q}} + t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}$$

Example: lw timing



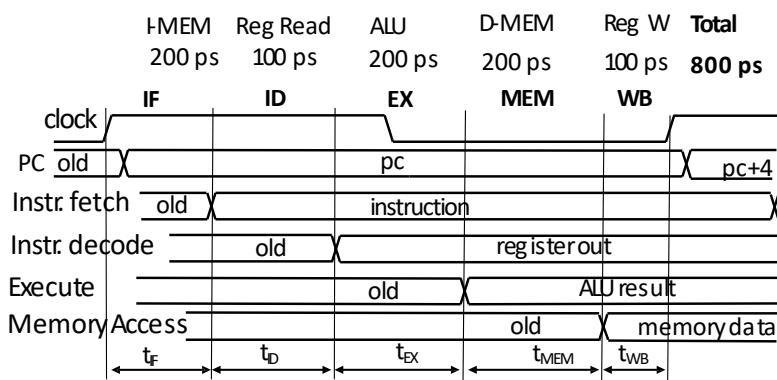
$$\text{Critical path} = t_{\text{clk-q}} + \max \{ t_{\text{Add}} + t_{\text{mux}}, t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{D MEM}} + t_{\text{mux}} \} + t_{\text{setup}}$$

8/27/2021

91 8/27/2021

92

Instruction Timing



8/27/2021

93

Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

- Maximum clock frequency
 - $f_{\text{max}} = 1/800\text{ps} = 1.25 \text{ GHz}$
- Most blocks idle most of the time
 - E.g. $f_{\text{max,ALU}} = 1/200\text{ps} = 5 \text{ GHz}$

94

Control Logic Design

8/27/2021

95

Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WSel	
add	*	*	+4		*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4		*	*	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4		*	*	Reg	Reg	(Op)	Read	1	ALU
addi	*	*	+4		I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4		I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4		S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4		B	*	PC	Imm	Add	Read	0	*
bne	1	*	ALU		B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU		B	*	PC	Imm	Add	Read	0	*
blt	*	1	ALU		B	0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU		B	1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU		I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU		J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4		U	*	PC	Imm	Add	Read	1	ALU

96

Control Realization Options

- ROM
 - “Read-Only Memory”
 - Regular structure
 - Can be easily reprogrammed
 - fix errors
 - add instructions
 - Popular when designing control logic manually
- Combinatorial Logic
 - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

RV32I, A Nine-Bit ISA!

imm[31:12]	rd	01001011111111111111111111111111	LUIPC
imm[31:12]	rd	01001111111111111111111111111111	JALR
imm[20:11] imm[11:9:2]	rd	01001111111111111111111111111111	JAL
imm[11:0]	rs1	000	imm[4:11]111111111111111111111111
imm[11:0]	rs2	001	imm[4:11]111111111111111111111111
imm[11:0]	rs1	000	imm[4:11]111111111111111111111111
imm[11:0]	rs2	000	imm[4:11]111111111111111111111111
imm[11:0]	rs1	010	imm[4:11]111111111111111111111111
imm[11:0]	rs2	010	imm[4:11]111111111111111111111111
imm[11:0]	rs1	111	imm[4:11]111111111111111111111111
imm[11:0]	rs2	100	imm[4:11]111111111111111111111111
imm[11:0]	rs1	001	rd
imm[11:0]	rs2	001	rd
imm[11:0]	rs1	000	rd
imm[11:0]	rs2	000	rd
imm[11:0]	rs1	101	rd
imm[11:0]	rs2	000	rd
imm[11:0]	rs1	011	rd
imm[11:0]	rs2	011	rd
imm[11:0]	rs1	010	imm[4:11]000000000000000000000000
imm[11:0]	rs2	010	imm[4:11]000000000000000000000000
imm[11:0]	rs1	010	imm[4:11]000000000000000000000000
imm[11:0]	rs2	010	imm[4:11]000000000000000000000000
imm[11:0]	rs1	011	rd
imm[11:0]	rs2	011	rd
imm[11:0]	rs1	010	SLTIU
imm[11:0]	rs2	010	XORI
imm[11:0]	rs1	010	OR
imm[11:0]	rs2	010	ANDI
imm[11:0]	rs1	111	SLII
imm[11:0]	rs2	111	SRII
shamt	rs1	001	SKAII
shamt	rs1	101	ADD
shamt	rs1	000	RD
shamt	rs2	001	SUB
shamt	rs2	101	SL
shamt	rs1	010	SLT
shamt	rs2	010	SLTU
shamt	rs1	011	XOR
shamt	rs2	011	SR
shamt	rs1	101	AND
shamt	rs2	101	FENCE
pred	rs1	000	ECALL
pred	rs2	000	EBREAK

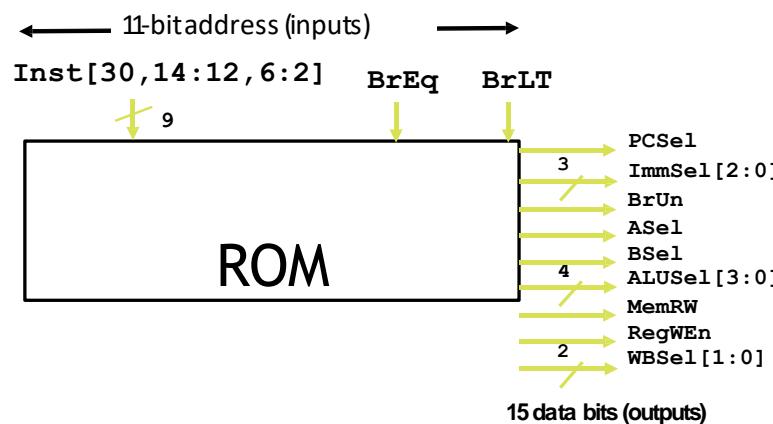
98

8/27/2021

97

8/27/2021

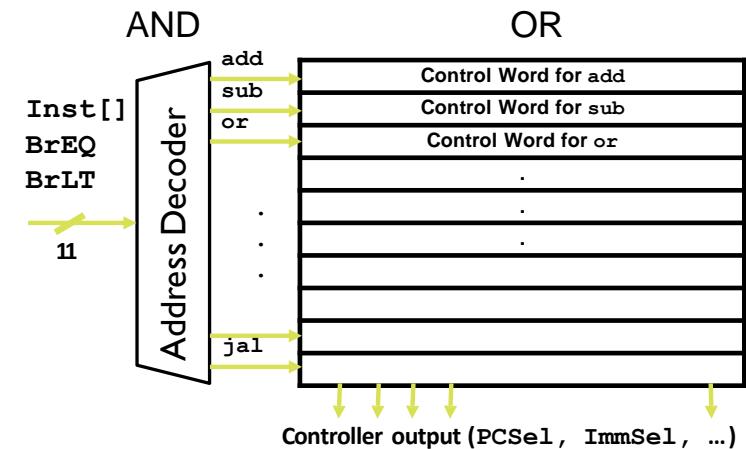
ROM-based Control



8/27/2021

99

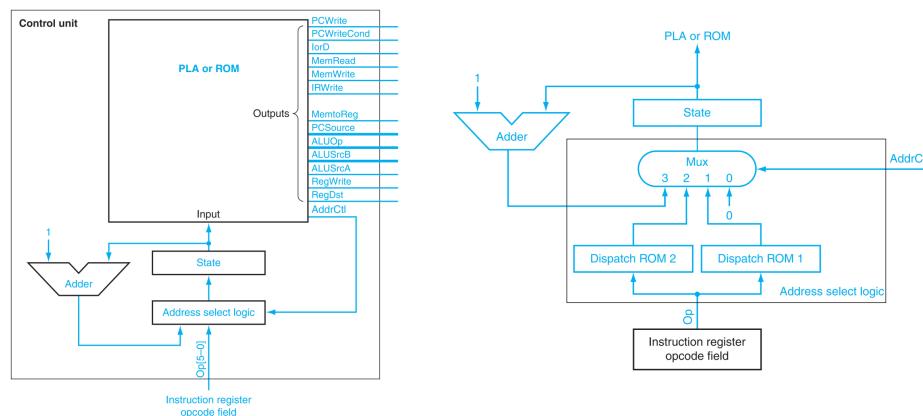
ROM Controller Implementation



8/27/2021

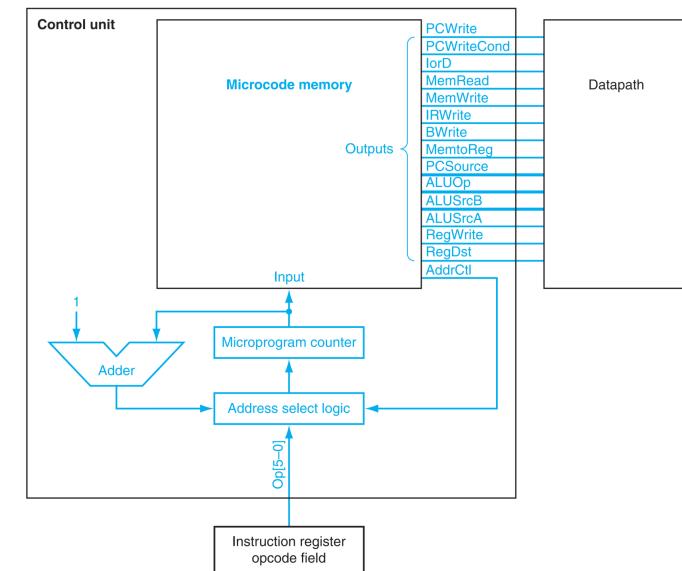
100

The control unit using an explicit counter to compute the next state



8/27/2021

101



8/27/2021

102

Combinational Logic Control

- Simplest example: BrUn

	inst[14:12]	inst[6:2]
imm[12]10:5	rs2	rs1
imm[12]10:5	rs2	rs1 000
imm[12]10:5	rs2	imm[4:1]111 110001 BEQ
imm[12]10:5	rs2	imm[4:1]111 110001 BNE
imm[12]10:5	rs2	imm[4:1]111 110001 BLT
imm[12]10:5	rs2	imm[4:1]111 110001 BGE
imm[12]10:5	rs2	imm[4:1]111 110001 BLTU
imm[12]10:5	rs2	imm[4:1]111 110001 BGEU

- How to decode whether BrUn is 1?

$$\text{BrUn} = \text{Inst}[13] \cdot \text{Branch}$$

Control Logic to Decode add

$$\text{add} = i[30] \cdot i[14] \cdot i[13] \cdot i[12] \cdot R\text{-type}$$

inst[30] inst[14:12] inst[6:2]

00000	shamt	rs1	001	rd	00100111	SLL
00000	shamt	rs1	101	rd	00100111	SRAL
01000	shamt	rs1	101	rd	00100111	SRAI
01000	rs2	rs1	000	rd	01100111	ADD
01000	rs2	rs1	000	rd	01100111	SUB
01000	rs2	rs1	001	rd	01100111	SLL
01000	rs2	rs1	010	rd	01100111	SLT
01000	rs2	rs1	011	rd	01100111	SLTU
01000	rs2	rs1	100	rd	01100111	XOR
01000	rs2	rs1	101	rd	01100111	SRL
01000	rs2	rs1	101	rd	01100111	SRA
01000	rs2	rs1	110	rd	01100111	OR
01000	rs2	rs1	111	rd	01100111	AND

$$R\text{-type} = \overline{i[6]} \cdot i[5] \cdot i[4] \cdot i[3] \cdot i[2] \cdot RV32I$$

$$RV32I = i[1] \cdot i[0]$$

8/27/2021

103

8/27/2021

104

“And In Conclusion...”

Call home, we've made HW/SW contact!

High Level Language
Program (e.g., C)

| Compiler

Assembly Language
Program (e.g., RISC-V)

| Assembler

Machine Language
Program (RISC-V)

Hardware Architecture Description
(e.g., block diagrams)

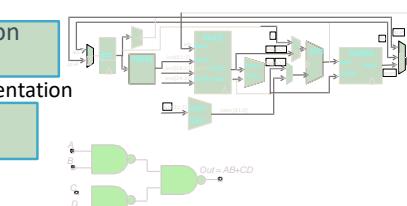
| Architecture Implementation

Logic Circuit Description
(Circuit Schematic Diagrams)

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw x3, 0(x10)
lw x4, 4(x10)
sw x4, 0(x10)
sw x3, 4(x10)

1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100



8/27/2021

105

8/27/2021

106

“And In conclusion...”

- We have built a processor!
 - Capable of executing all RISC-V instructions in one cycle each
 - Not all units (hardware) used by all instructions
 - Critical path changes
- 5 Phases of execution
 - IF, ID, EX, MEM, WB
 - Not all instructions are active in all phases
- Controller specifies how to execute instructions
 - Implemented as ROM or logic

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

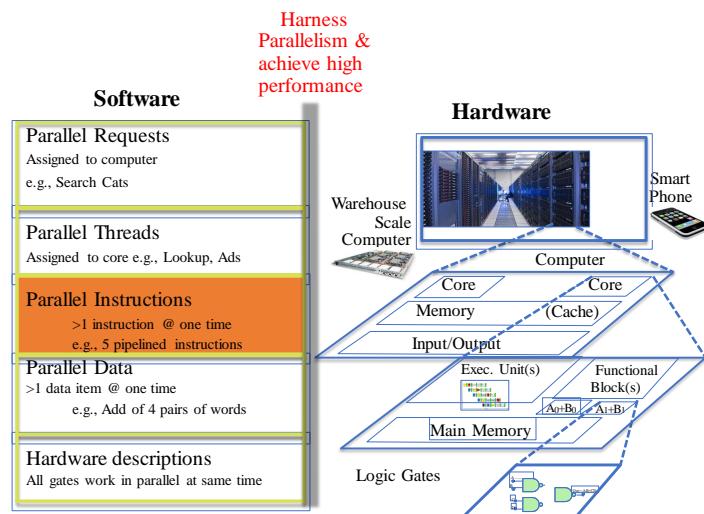
[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great Ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolic, © 2020, UC Berkeley]

8/27/2021

107

1

New-School Machine Structures



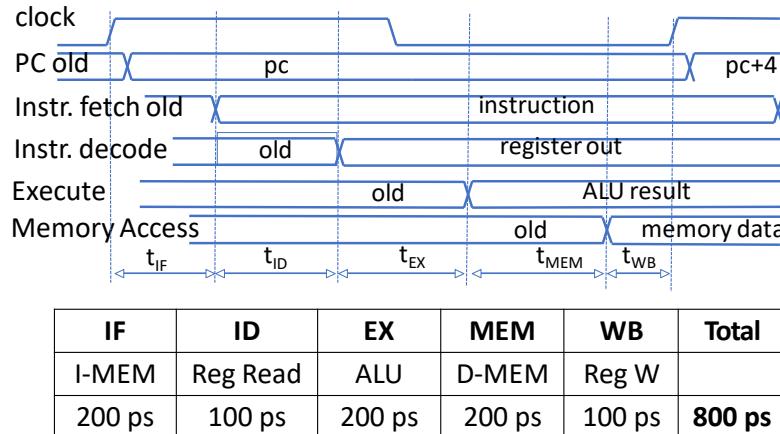
6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. **Parallelism**
5. Performance Measurement & Improvement
6. Dependability via Redundancy

2

3

Instruction Timing



Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

- Maximum clock frequency

$$\square f_{\max} = 1/800\text{ps} = 1.25 \text{ GHz}$$

Performance Measures

- “Our” Single-cycle RISC-V CPU executes instructions at 1.25 GHz
 - 1 instruction every 800 ps
- Can we improve its performance?
 - What do we mean with this statement?
 - Not so obvious:
 - Quicker response time, so one job finishes faster?
 - More jobs per unit time (e.g. web server returning pages, spoken words recognized)?
 - Longer battery life?

Transportation Analogy



	Sports Car	Bus
Passenger Capacity	2	50
Travel Speed	200 mph	50 mph
Gas Mileage	5 mpg	2 mpg

50 Mile trip (assume they return instantaneously)

	Sports Car	Bus
Travel Time	15 min	60 min
Time for 100 passengers	750 min (50 2-person trips)	120 min (two 50-person trips)
Gallons per passenger	5 gallons	0.5 gallons

Computer Analogy

Transportation	Computer
Trip Time	Program execution time: e.g. time to update display
Time for 100 passengers	Throughput: e.g. number of server requests handled per hour
Gallons per passenger	Energy per task*: e.g. how many movies you can watch per battery charge or energy bill for datacenter

* Note: Power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time

RISC-V

8

Processor Performance Iron Law

9

“Iron Law” of Processor Performance

$$\frac{\text{Time}_{\text{Program}}}{\text{Instructions}_{\text{Program}}} = \frac{\text{Instructions}_{\text{Program}}}{\text{Instructions}_{\text{Instruction}}} * \frac{\text{Cycles}_{\text{Instruction}}}{\text{Cycles}_{\text{Instruction}}} * \frac{\text{Time}_{\text{Cycle}}}{\text{Time}_{\text{Cycle}}}$$

↑

CPI = Cycles Per Instruction

Defining (Speed) Performance

- Normally interested in reducing
 - **Response time** (aka execution time) – the time between the start and the completion of a task
 - Important to individual users
 - Thus, to maximize performance, need to **minimize** execution time

$$\text{performance}_X = 1 / \text{execution_time}_X$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution_time}_Y}{\text{execution_time}_X} = n$$

- Throughput – the total amount of work done in a given time
 - Important to data center managers
- Decreasing response time almost always improves throughput

RISC-V

10

Performance Factors

- Want to distinguish elapsed time and the time spent on our task
- CPU execution time (CPU time) – time the CPU spends working on a task
 - Does not include time waiting for I/O or running other programs

$$\text{CPU execution time} = \frac{\# \text{ CPU clock cycles for a program}}{\text{clock cycle time for a program}}$$

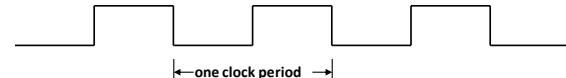
$$\text{CPU execution time} = \frac{\# \text{ CPU clock cycles for a program}}{\text{clock rate}}$$

- Can improve performance by reducing either the **length of the clock cycle** or the **number of clock cycles required for a program**

Review: Machine Clock Rate

- Clock rate (MHz, GHz) is inverse of clock cycle time (clock period)

$$CC = 1 / CR$$



10 nsec clock cycle => 100 MHz clock rate

5 nsec clock cycle => 200 MHz clock rate

2 nsec clock cycle => 500 MHz clock rate

1 nsec clock cycle => 1 GHz clock rate

500 psec clock cycle => 2 GHz clock rate

250 psec clock cycle => 4 GHz clock rate

200 psec clock cycle => 5 GHz clock rate

Clock Cycles per Instruction

- Not all instructions take the same amount of time to execute
 - One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction

$$\# \text{ CPU clock cycles for a program} = \# \text{ Instructions for a program} \times \text{Average clock cycles per instruction}$$

- Clock cycles per instruction (CPI)** – the average number of clock cycles each instruction takes to execute
 - A way to compare two different implementations of the same ISA

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

Effective CPI

- Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging

$$\text{Overall effective CPI} = \sum_{i=1}^n (\text{CPI}_i \times IC_i)$$

Where IC_i is the count (percentage) of the number of instructions of class i executed

CPI_i is the (average) number of clock cycles per instruction for that instruction class

n is the number of instruction classes

- The overall effective CPI varies by instruction mix – a measure of the dynamic frequency of instructions across one or many programs

THE Performance Equation

- Our basic performance equation is then

$$\text{CPU time} = \text{Instruction_count} \times \text{CPI} \times \text{clock_cycle}$$

or

$$\text{CPU time} = \frac{\text{Instruction_count} \times \text{CPI}}{\text{clock_rate}}$$

- These equations separate the **three key** factors that affect performance
 - Can measure the CPU execution time by running the program
 - The clock rate is usually given
 - Can measure overall instruction count by using profilers/ simulators without knowing all of the implementation details
 - CPI varies by instruction type and ISA implementation for which we must know the implementation details

A Simple Example

Op	Freq	CPI _i	Freq x CPI _i	.5	.5	.25
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			$\Sigma = 2.2$	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

CPU time new = $1.6 \times IC \times CC$ so $2.2/1.6$ means 37.5% faster

- How does this compare with using branch prediction to shave a cycle off the branch time?

CPU time new = $2.0 \times IC \times CC$ so $2.2/2.0$ means 10% faster

- What if two ALU instructions could be executed at once?

CPU time new = $1.95 \times IC \times CC$ so $2.2/1.95$ means 12.8% faster

Instructions per Program

Determined by

- Task
- Algorithm, e.g. $O(N^2)$ vs $O(N)$
- Programming language
- Compiler
- Instruction Set Architecture (ISA)

(Average) Clock Cycles per Instruction (CPI)

Determined by

- ISA
- Processor implementation (or *microarchitecture*)
- E.g. for “our” single-cycle RISC-V design, CPI = 1
- Complex instructions (e.g. **strcpy**), CPI $>> 1$
- Superscalar processors, CPI < 1 (next lectures)

Time per Cycle (1/Frequency)

Determined by

- Processor microarchitecture (determines critical path through logic gates)
- Technology (e.g. 5nm versus 28nm)
- Power budget (lower voltages reduce transistor speed)

RISC-V

20

Speed Tradeoff Example

- For some task (e.g. image compression) ...

	Processor A	Processor B
# Instructions	1 Million	1.5 Million
Average CPI	2.5	1
Clock rate f	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

Processor B is faster for this task, despite executing more instructions and having a slower clock rate!

RISC-V

21

Determinates of CPU Performance

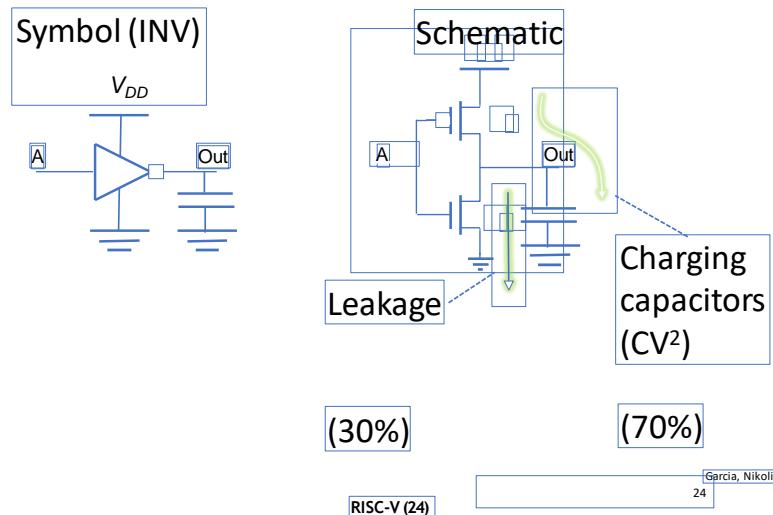
$$\text{CPU time} = \text{Instruction_count} \times \text{CPI} \times \text{clock_cycle}$$

	Instruction_count	CPI	clock_cycle
Algorithm	x	x	
Programming language	x	x	
Compiler	x	x	
ISA	x	x	x
Processor organization		x	x
Technology			x

Energy
Efficiency

23

Where Does Energy Go in CMOS?



Energy per Task

$$\text{Energy} = \frac{\text{Instructions Program}}{\text{Program}} * \text{Energy Instruction}$$

$$\text{Energy} \propto \frac{\text{Instructions Program}}{\text{Program}} * C V^2$$

"Capacitance" depends on technology, processor features e.g. # of cores

Supply voltage, e.g. 1V

Want to reduce capacitance and voltage to reduce energy/task

RISC-V

25

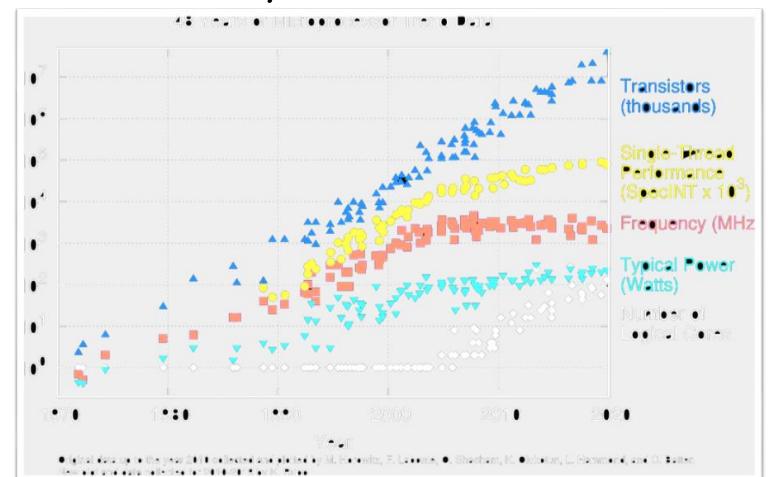
Energy Tradeoff Example

- “Next-generation” processor
 - C (Moore’s Law): -15 %
 - Supply voltage, V_{sup} : -15 %
 - Energy consumption: $0 - (1 - 0.85^3) = -39 \%$
- Significantly improved energy efficiency thanks to
 - Moore’s Law AND
 - Reduced supply voltage

RISC-V (26)

26

Performance/Power Trends



RISC-V

27

End of Scaling

- In recent years, industry has not been able to reduce supply voltage much, as reducing it further would mean increasing “leakage power” where transistor switches don’t fully turn off (more like dimmer switch than on-off switch)
- Also, size of transistors and hence capacitance, not shrinking as much as before between transistor generations
 - Need to go to 3D
- Power becomes a growing concern – the “power wall”

RISC-V (28)

28

Energy “Iron Law”

- $$\text{Performance} = \frac{\text{Power}}{\text{(Tasks/Second)}} * \frac{\text{Energy Efficiency}}{\text{(Joules/Second)}} = \frac{\text{Tasks}}{\text{Joule}}$$
- Energy efficiency (e.g., instructions/Joule) is key metric in all computing devices
 - For power-constrained systems (e.g., 20MW datacenter), need better energy efficiency to get more performance at same power
 - For energy-constrained systems (e.g., 1W phone), need better energy efficiency to prolong battery life

RISC-V (21)

29

Introduction to Pipelining

30

Gotta Do Laundry

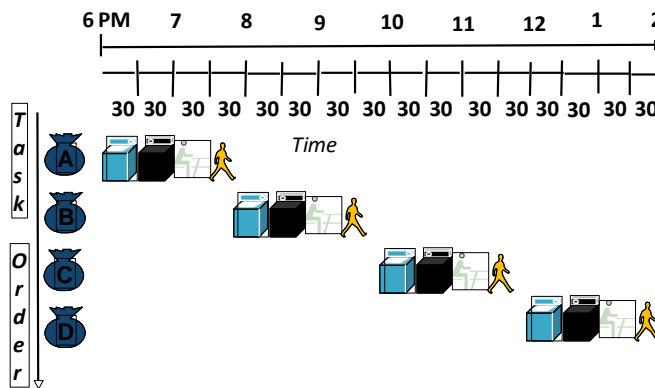
- Avi, Bora, Caroline, Dan each have one load of clothes to wash, dry, fold, and put away
 - Washer takes 30 minutes
 - Dryer takes 30 minutes
 - “Folder” takes 30 minutes
 - “Stasher” takes 30 minutes to put clothes into drawers



RISC-V

31

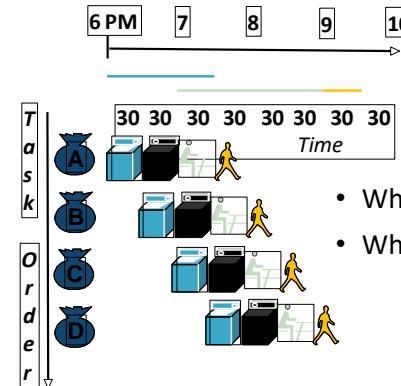
Sequential Laundry



Sequential laundry takes 8 hours for 4 loads!

RISC-V (32)

Pipelined Laundry

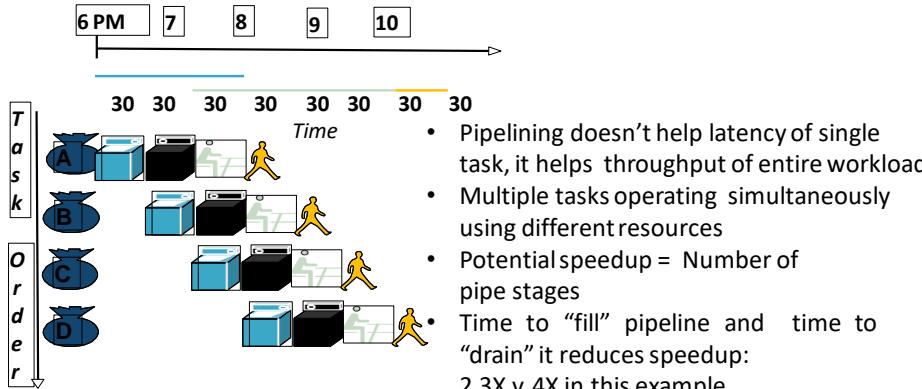


- What happens *sequentially*?
- What happens *simultaneously*?

Pipelined laundry takes 3.5 hours for 4 loads!

RISC-V

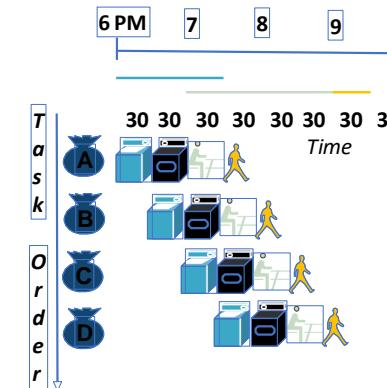
Sequential Laundry



RISC-V

32

Sequential Laundry



Suppose:
 - new Washer takes 20 minutes
 - new Stasher takes 20 minutes.

How much faster is pipeline?

Pipeline rate limited by slowest pipeline stage

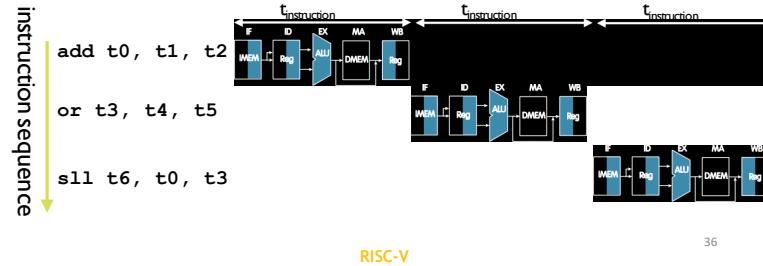
Unbalanced lengths of pipe stages reduce speedup

RISC-V

33

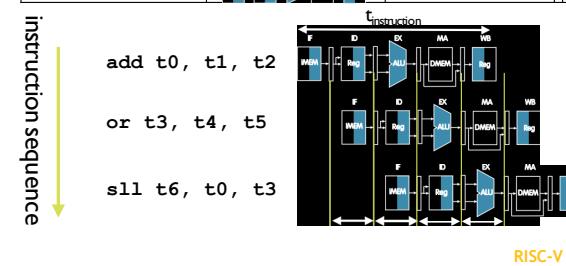
'Sequential' RISC-V Datapath

Phase	Pictogram	t_{step} Serial
Instruction Fetch		200 ps
Reg Read		100 ps
ALU		200 ps
Memory		200 ps
Register Write		100 ps
$t_{instruction}$		800 ps

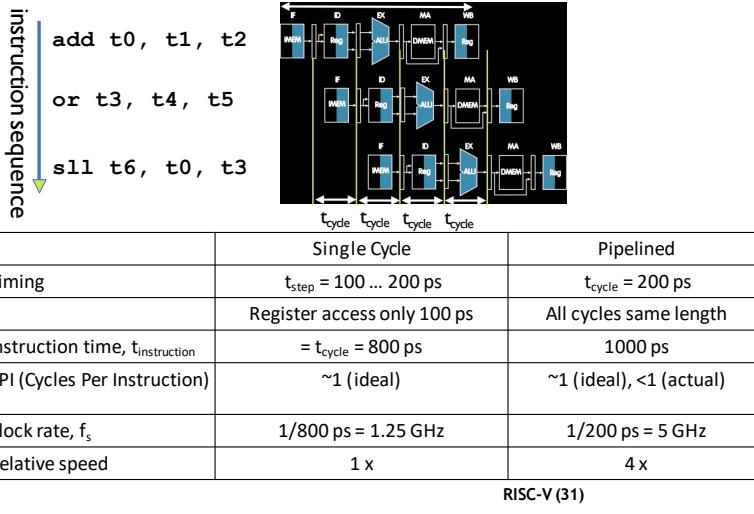


Pipelined RISC-V Datapath

Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps
Reg Read		100 ps	200 ps
ALU		200 ps	200 ps
Memory		200 ps	200 ps
Register Write		100 ps	200 ps
$t_{instruction}$		800 ps	1000 ps

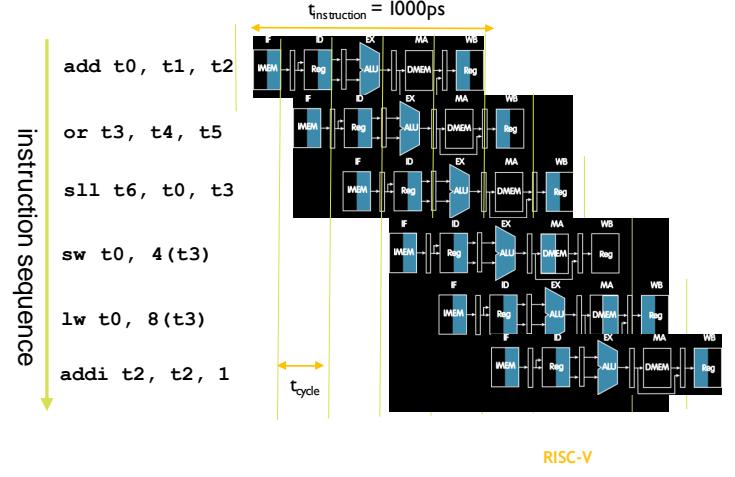


Pipelined RISC-V Datapath



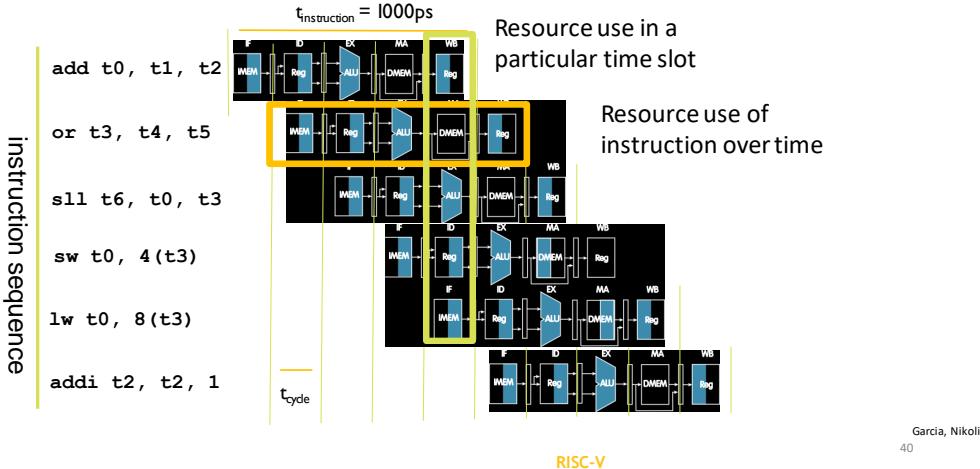
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?



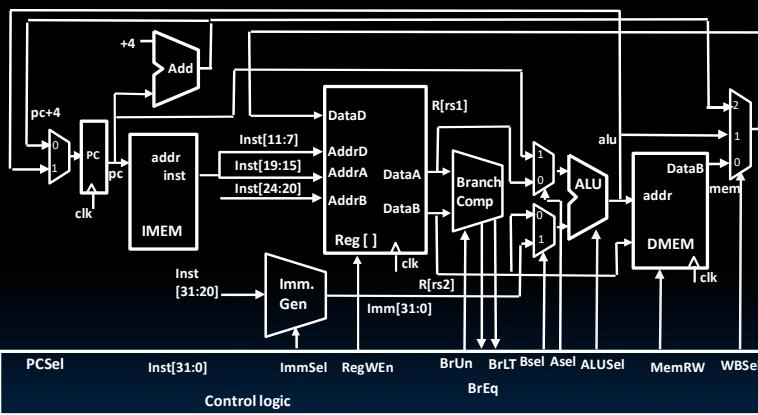
Sequential vs. Simultaneous

- What happens sequentially and what simultaneously?

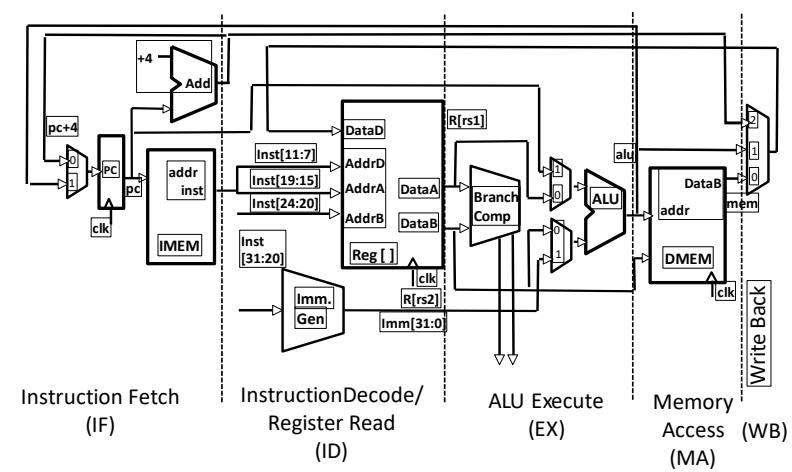


Pipelining Datapath

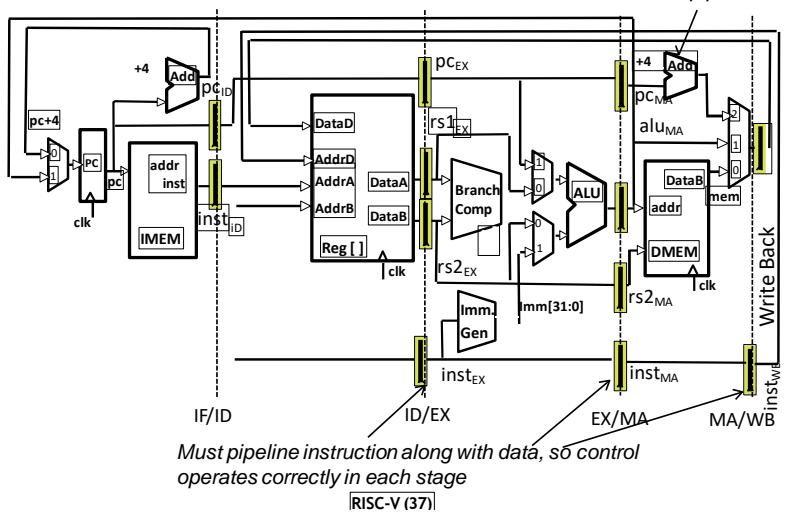
Single-Cycle



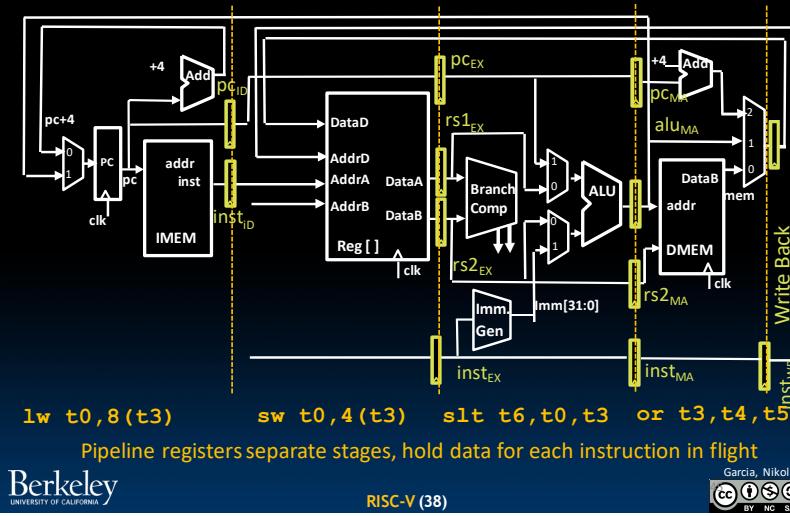
Single-Cycle RV32I Datapath



Pipelined RV32I Datapath



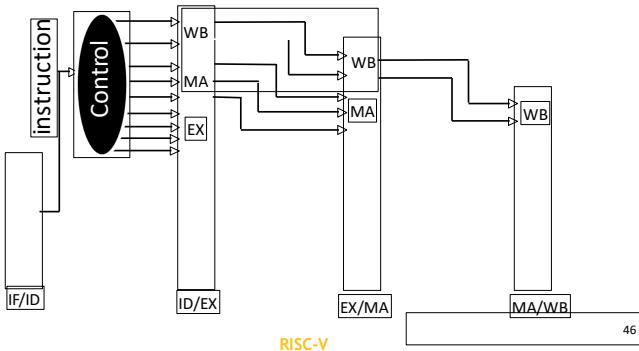
Pipelined RV32I Datapath



44

Pipelined Control

- Control signals derived from instruction
 - As in single-cycle implementation
 - Information is stored in pipeline registers for use by later stages



Pipeline Hazards

47

Hazards Ahead!



RISC-V

RISC-V



RISC-V

49

Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) Structural hazard

- A required resource is busy (e.g. needed in multiple stages)

2) Data hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

3) Control hazard

- Flow of execution depends on previous instruction

Structural Hazard

- **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource
- **Solution 1:** Instructions take it in turns to use resource, some instructions have to stall
- **Solution 2:** Add more hardware to machine
- Can always solve a structural hazard by adding more hardware

Regfile Structural Hazards

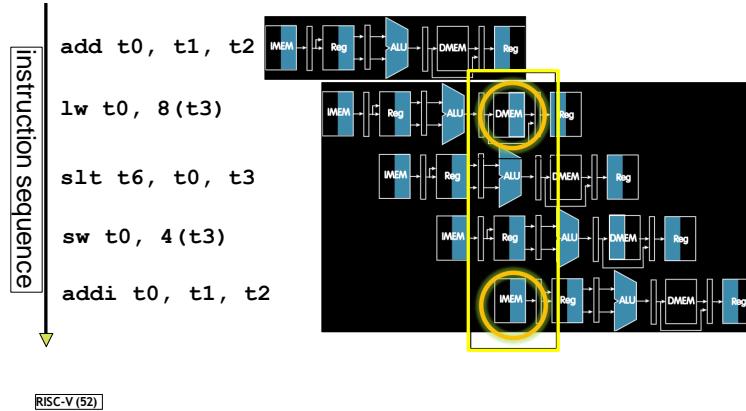
- Each instruction:
 - Can read up to two operands in decode stage
 - Can write one value in writeback stage
- Avoid structural hazard by having separate “ports”
 - Two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

RISC-V

50

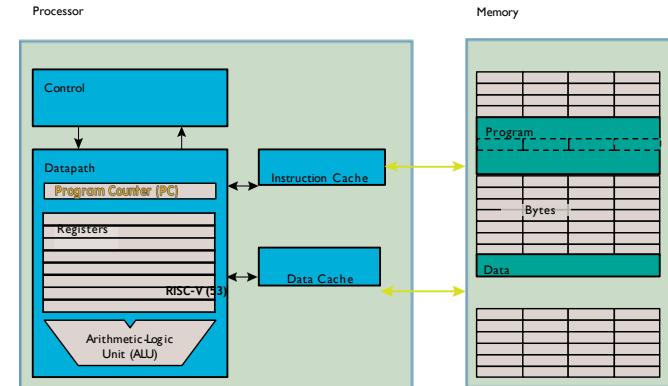
Structural Hazard: Memory Access

- Instruction and data memory used simultaneously
 - ✓ Use two separate memories



Instruction and Data Caches

- Fast, on-chip memory, separate for instructions and data



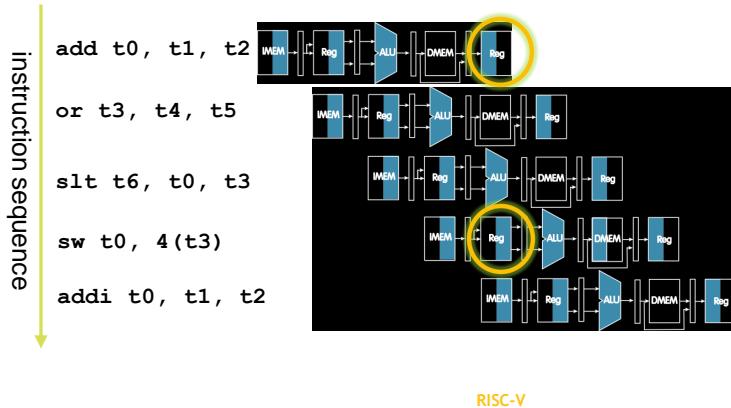
Structural Hazards – Summary

- Conflict for use of a resource
- In RISC-V pipeline with a single memory
 - Load/store requires data access
 - Without separate memories, instruction fetch would have to stall for that cycle
 - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memories
 - Or separate instruction/data caches
- RISC ISAs (including RISC-V) designed to avoid structural hazards
 - e.g. at most one memory access/instruction

**Data
Hazards**

Data Hazard: Register Access

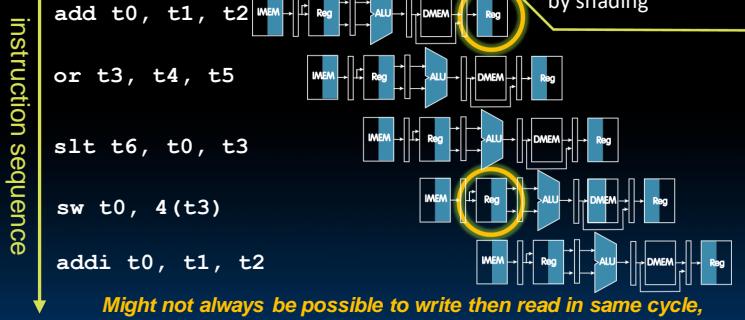
- Separate ports, but what if write to same register as read?
- Does **sw** in the example fetch the old or new value?



56



- Exploit high speed of register file (100 ps)
 - WB updates value
 - ID reads new value
- Indicated in diagram by shading



Berkeley
UNIVERSITY OF CALIFORNIA

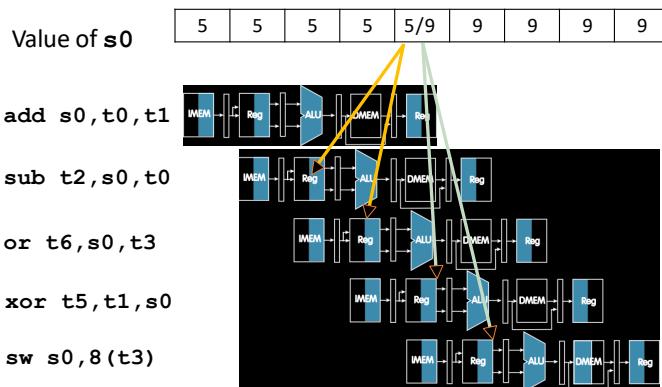
Might not always be possible to write then read in same cycle,
especially in high-frequency designs. Check assumptions in Garcia, Nikolic

RISC-V (50)



Data Hazard: ALU Result

instruction sequence



Without some fix, **sub** and **or** will calculate wrong result!

RISC-V (51)

58

Solution 1: Stalling

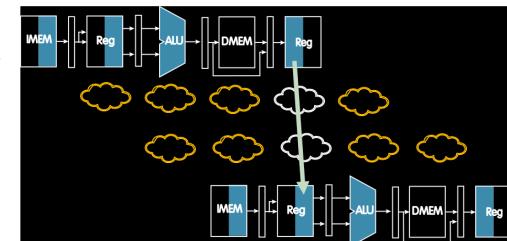
- Problem: Instruction depends on result from previous instruction

add s0, t0, t1
sub t2, s0, t3

add s0, t0, t1
"bubbles"

sub t2, s0, t3

- Bubble:
 - Effectively **nop**: Affected pipeline stages do "nothing"



RISC-V

59

Stalls and Performance

- Stalls reduce performance
 - But stalls are required to get correct results
- Compiler can arrange code or insert **nops** (`addi x0, x0, 0`) to avoid hazards and stalls
 - Requires knowledge of the pipeline structure

RISC-V

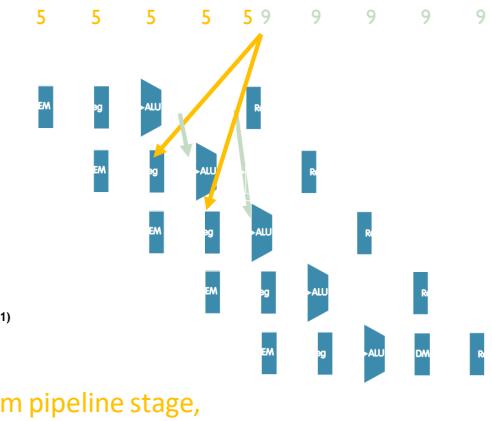
60

Solution 2: Forwarding

- Value of s_0
- `add s0, t0, t1` `sub t2, s0, t0`
- `or t6, s0, t3`
- `xor t5, t1, s0` `sw s0, 8(t3)`

• Forwarding: grab operand from pipeline stage,

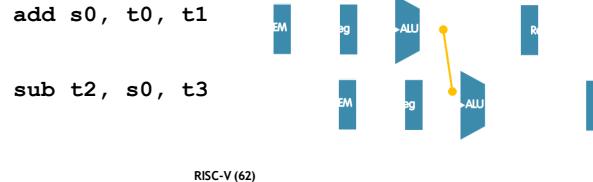
rather than register file



Garcia, Nikolić
CC BY NC SA

Forwarding (aka Bypassing)

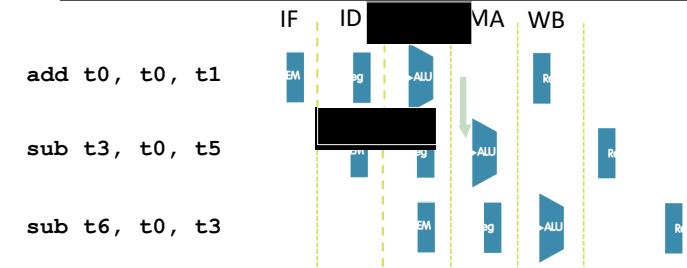
- Use result when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath



Garcia, Nikolić

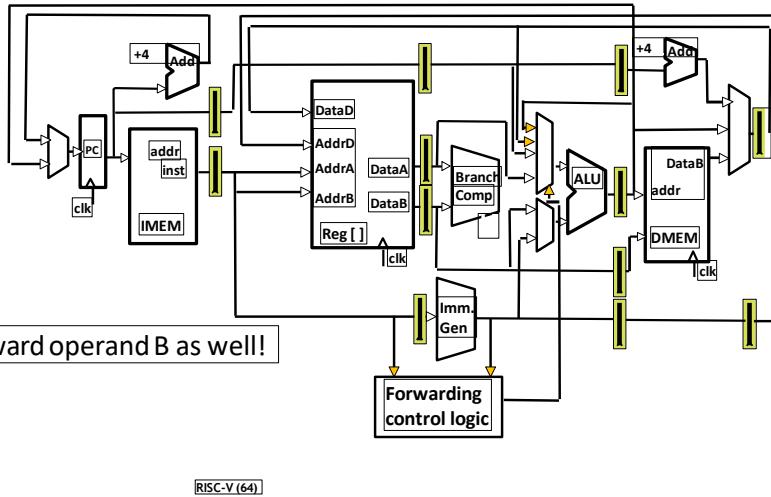
Data Needed for Forwarding (Example)

- Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
- Must ignore writes to x_0 !



RISC-V (63)

Pipelined RV32I Datapath



Remember to forward operand B as well!

RISC-V (64)

Data Hazard and Forwarding

- Value of s_0

5 5 5 5 5/9 9 9 9 9

- $\text{add } s_0, t_0, t_1$

IMEM Reg ALU DMEM Reg

- $\text{sub } t_2, s_0, t_0$

IMEM Reg ALU DMEM Reg

- $\text{or } t_6, s_0, t_3$

IMEM Reg ALU DMEM Reg

- $\text{xor } t_5, t_1, s_0$

IMEM Reg ALU DMEM Reg

- $\text{sw } s_0, 8(t_3)$

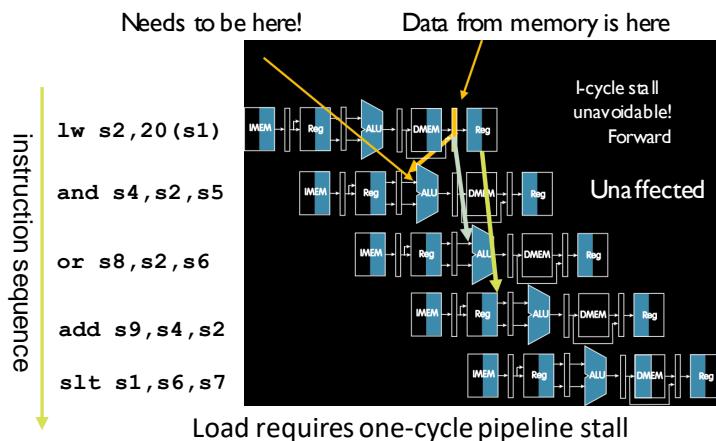
IMEM Reg ALU DMEM Reg

- Forwarding: grab operand from pipeline stage, rather than register file

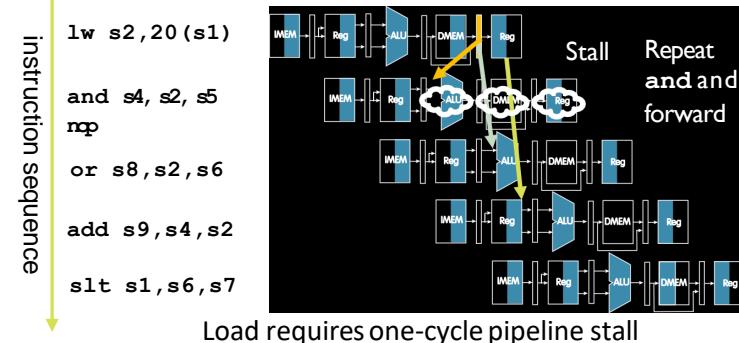
RISC-V (59)

65

Load Data Hazard



Stall Pipeline



Load requires one-cycle pipeline stall

RISC-V

66

RISC-V (61)

67

1w Data Hazard

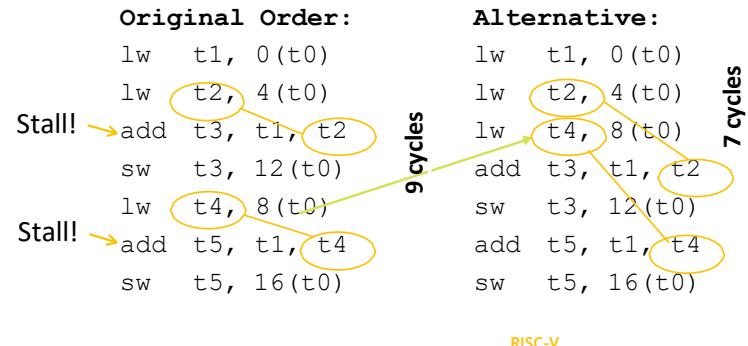
- Slot after a load is called a *load delay slot*
 - If that instruction uses the result of the load, then the hardware will stall for one cycle
 - Equivalent to inserting an explicit **nop** in the slot
 - except the latter uses more code space
 - Performance loss
- Idea:
 - Put unrelated instruction into load delay slot
 - No performance loss!

RISC-V

68

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instr!
- RISC-V code for **A[3]=A[0]+A[1]; A[4]=A[0]+A[2]**

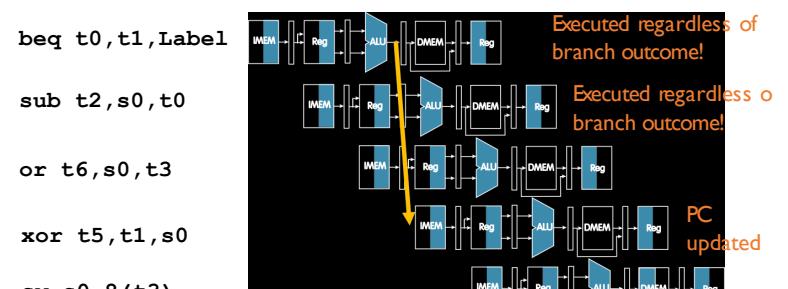


69

Control Hazards

70

Control Hazards



RISC-V

71

Observation

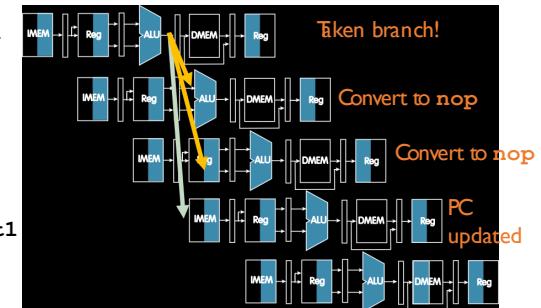
- If branch not taken, then instructions fetched sequentially after branch are correct
- If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs

RISC-V

72

Kill Instructions after Branch if Taken

```
beq t0,t1,Label  
sub t2,s0,t0  
or t6,s0,t3  
Label: xor t5,t1
```



RISC-V

73

Reducing Branch Penalties

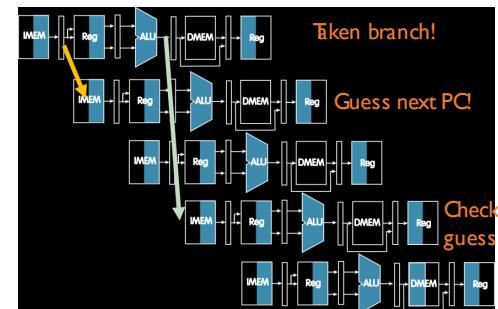
- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect

RISC-V

74

Branch Prediction

```
beq t0,t1,Label  
Label :...
```



RISC-V

75

Superscalar Processors

Increasing Processor Performance

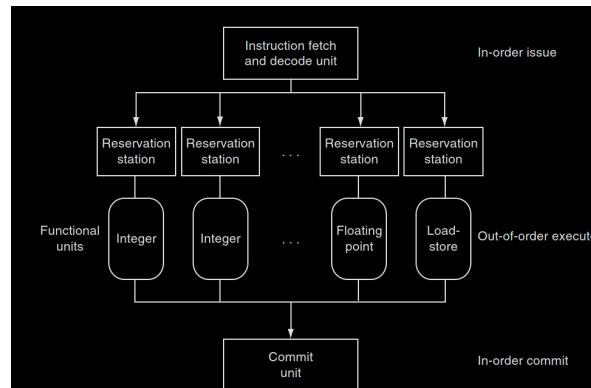
1. Clock rate
 - Limited by technology and power dissipation
2. Pipelining
 - “Overlap” instruction execution
 - Deeper pipeline: 5 => 10 => 15 stages
 - Less work per stage → shorter clock cycle
 - But more potential for hazards (CPI > 1)
3. Multi-issue “superscalar” processor

76

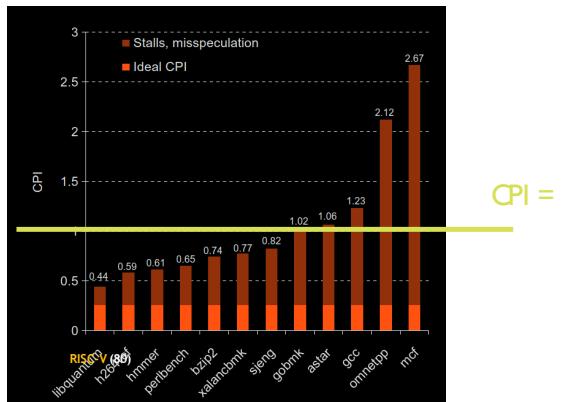
Superscalar Processor

- Multiple issue “superscalar”
 - Replicate pipeline stages => multiple pipelines
 - Start multiple instructions per clock cycle
 - CPI < 1, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak CPI = 0.25, peak IPC = 4
 - Dependencies reduce this in practice
- “Out-of-Order” execution
 - Reorder instructions dynamically in hardware to reduce impact of hazards
- *CS152 discusses these techniques!*

Superscalar Processor



Benchmark: CPI of i7



“Iron Law” of Processor Performance



CPI = Cycles Per Instruction

Can time Can count Can look up

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$



RISC-V (81)

$$\text{CPI} = \frac{\text{Cycles}}{\text{Instruction}} = \frac{\text{Time}}{\text{Program}} \div \left(\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}} \right)$$

Berkeley
UNIVERSITY OF CALIFORNIA



Pipelining and ISA Design

- RISC-V ISA designed for pipelining
 - All instructions are 32-bits
 - Easy to fetch and decode in one cycle
 - Versus x86: 1- to 15-byte instructions
 - Few and regular instruction formats
 - Decode and read registers in one step
 - Load/store addressing
 - Calculate address in 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - Memory access takes only one cycle

RISC-V (82)

“And In conclusion...”

- We have built a processor!
 - Capable of executing all RISC-V instructions in one cycle each
- 5 Phases of execution
 - IF, ID, EX, MEM, WB
 - Not all instructions are active in all phases
- Controller specifies how to execute instructions
 - Implemented as ROM or logic
- Pipelining improves performance
 - But we must resolve hazards

RISC-V (83)

Computer Architecture 1

Computer Organization and Design
THE HARDWARE/SOFTWARE INTERFACE

Binary Prefix

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great Ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolic, © 2020, UC Berkeley]

1

Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

- Common use prefixes (all SI, except K [= k in SI])
- Confusing! Common usage of “kilobyte” means 1024 bytes, but the “correct” SI value is 1000 bytes
- Hard Disk manufacturers & Telecommunications are the only computing groups that use SI factors
 - What is advertised as a 1 TB drive actually holds about 90% of what you expect
 - A 1 Mbit/s connection transfers 10^6 bps.

Name	Abbr	Factor	SI size
Kilo	K	$2^{10} = 1024$	$10^3 = 1,000$
Mega	M	$2^{20} = 1048,576$	$10^6 = 1,000,000$
Giga	G	$2^{30} = 1,073,741,824$	$10^9 = 1,000,000,000$
Tera	T	$2^{40} = 1,099,511,627,776$	$10^{12} = 1,000,000,000,000$
Peta	P	$2^{50} = 1,125,899,906,842,624$	$10^{15} = 1,000,000,000,000,000$
Exa	E	$2^{60} = 1,152,921,504,606,846,976$	$10^{18} = 1,000,000,000,000,000,000$
Zetta	Z	$2^{70} = 1,180,591,620,717,411,303,424$	$10^{21} = 1,000,000,000,000,000,000,000$
Yotta	Y	$2^{80} = 1,208,925,819,614,629,174,706,176$	$10^{24} = 1,000,000,000,000,000,000,000,000$

Caches I

kibi, mebi, gibi, tebi, pebi, exbi, zebi, yobi

- IEC Standard Prefixes [only to exbi officially]

Name	Abbr	Factor
kibi	Ki	$2^{10} = 1024$
mebi	Mi	$2^{20} = 1048,576$
gibi	Gi	$2^{30} = 1,073,741,824$
tebi	Ti	$2^{40} = 1,099,511,627,776$
pebi	Pi	$2^{50} = 1,125,899,906,842,624$
exbi	Ei	$2^{60} = 1,152,921,504,606,846,976$
zebi	Zi	$2^{70} = 1,180,591,620,717,411,303,424$
yobi	Yi	$2^{80} = 1,208,925,819,614,629,174,706,176$

- International Electrotechnical Commission (IEC) in 1999
- Names come from shortened versions of the original SI prefixes (same pronunciation) and bi is short for “binary”, but pronounced “bee” :-)
- Now SI prefixes only have their base-10 meaning and never have a base-2 meaning

Caches I