PURDUE UNIVERSITY

CS 25200: Systems Programming

Lecture 16: Wildcards, Computer Architecture

Prof. Turkstra

Lecture 16

- Wildcards
- Computer architecture review
- DMA

Shell

Final Command Table

ls	-al	aab	aaa
grep	me		
In:dflt	Out:file1	Err:dflt	

Lexer

Parser

shell.l → shell.y → wildcards env vars → executor

ls -al a* | grep me > file1

<ls> <-al>
<a*> <PIPE>
<grep> <me>
<GREAT>
<file1>

Command Table

ls	-al	a*
grep	me	
In:dflt	Out:file1	Err:dflt

Wildcards

- Allow us to perform actions on more than one file at a time
- Also called file globbing
- Do not confuse with regular expressions

Standard wildcards

- $ man 7 glob
  - ?: any single character
    - DSCN00??.JPG
  - *: any number of characters, including zero
    - a*a *.c t*.log
  - [ ]: a range
    - m[a-d]m m[0-7]n
  - { }: “or” relationship – no spaces allowed
    - cp {*.doc,*.pdf} ~
  - [!]: anything not in the range
  - \: escape character

Our shell

- Only need to worry about * and ?
- Implement the simple case first
  - Add a function to shell.y to do the expansion

shell.y

- Before...

```
argument: WORD {
  insert_single_command(current_command, $1);
} ;
```

- After...

```
argument: WORD {
  expand_wildcards($1);
} ;
```

POSIX Regular expressions

- int regcomp(regex_t *preg, const char *regex,
  int cflags)
  - Compiles the regular expression into a form that regexec() can use
  - Completes the passed regex_t
    - Finite state automata
- int regexec(const regex_t *preg, const char *string,
  size_t nmatch, regmatch_t pmatch[], int eflags)
  - Do the actual comparison

```
typedef struct {
        regoff_t rm_so;
        regoff_t rm_eo;
    } regmatch_t;
```

void regfree(regex_t *preg)

- regex_t's contain dynamically allocated memory
- You have to free it
  - ...or you'll have a memory leak
- Even have to free before reusing!

- Does not free the regex_t itself

Implementation

- Simple case...

```
void expand_wildcards(char *arg) {
  // return if arg does not contain wildcard(s)
  if (arg has neither '*' or '?') {
    // strchr() might help
    insert_argument(arg);
    return;
  }
```

Implementation

```
// 1. Convert wildcard to regular expression
// Convert “*” -> “.*”
// “?” -> “.”
// “.” -> “\.” and others you need
// Also add ^ at the beginning and $ at the end to match
// the beginning ant the end of the word.
// Allocate enough space for regular expression
  char *regex = (char *) malloc(2 * strlen(arg) + 3);
  char *arg_pos = arg;
  char *regex_pos = regex;

  *regex_pos++ = '^'; // match beginning of line
  while (*arg_pos) {
    if (*arg_pos == '*') {
      *regex_pos++ = '.';
      *regex_pos++ = '*';
    }
```

Implementation

```
  else if (*arg_pos == '?')
    *regex_pos++ = '.';
  }
  else if (*arg_pos == '.') {
    *regex_pos++ = '\\';
    *regex_pos++ = '.';
  }
  else {
    *rex_pos++ = *arg_pos;
  }
  arg_pos++;
}
*regex_pos++ = '$';
*regex_pos = '\0'; // match end of line and add null char
```

Implementation

```
  // 2. compile regular expression. See regular.c example
  regex_t re;
  int status = regcomp(&re, regex, ... );
  if (status != 0) {
    perror("compile");
    return;
  }

  // 3. List directory and add as arguments the entries
  // that match the regular expression
  DIR *dir = opendir(".");
  if (dir == NULL) {
    perror("opendir");
    return;
  }
  struct dirent *ent;

  while ((ent = readdir(dir)) != NULL) {
    // Check if name matches
    regmatch_t match;
    if (regexec(&re, ent->d_name, ...) == 0) {
      // Add argument
      insert_argument(strdup(ent->d_name));
    }
  }
  closedir(dir);
}
```

Note!

- Previous code is incomplete and contains errors
- Only intended as a starting point

Sorting

- Entries generated from a wildcard are presented in sorted order by most shells
  - E.g., $ echo *
- Your shell should too

Sorting

```
struct dirent *ent;
int max_entries = 20;
int num_entries = 0;
char **array = (char **) malloc(max_entries * sizeof(char *));
while ((ent = readdir(dir)) != NULL) {
  // Check if name matches
  if (regexec(ent->d_name, exp_buf)) {
    if (num_entries == max_entries) {
      max_entries *= 2;
      array = realloc(array, max_entries * sizeof(char *));
      assert(array != NULL);
    }
    array[num_entries]= strdup(ent->d_name);
    num_entries++;
  }
}
closedir(dir);
sort_array_strings(array, num_entries);
// Add arguments
for (int i = 0; i < num_entries; i++) {
  insert_argument(strdup(array[i]));
}
free(array); // What's wrong here?
```

Dot files

- Sometimes called “hidden” files
  - ., .., .login, .bashrc, etc
- Not included in directory listings
  - Unless -a
- Not matched by wildcards
  - Unless first character of pattern is .

```
if (regexec (...) ) {
  if (ent->d_name[0] == '.') {
    if (arg[0] == '.')
      add filename to arguments;
    }
  }
  else {
    add ent->d_name to arguments
  }
}
```

Subdirectory wildcards

- Wildcards match directories in a pattern as well
  - E.g. echo /e*/a*/b*/*.conf

Rabbit holes

- /u*/l*/a*
  - /usr/l*/a*
    - /usr/lib/a*
      - /usr/lib/aa
      - /usr/lib/abb
    - /usr/local/a*
      - /usr/local/axz
      - /usr/local/a12
  - /u/l*/a*
    - /u/loc/a*
      - /u/loc/aq
      - /u/loc/ar
    - /u/lll/a*
      - /u/lll/apple
      - /u/lll/atp
  - /unix/l*/a*
    - /unix/134/a*
      - /unix/134/abcd
      - /unix/134/a45
    - /unix/lll/a*
      - /unix/lll/ab
      - /unix/lll/ap3

Subdirectory wildcards

- One approach, write a function: `expand_wildcards(prefix, suffix)`
  - prefix: expanded portion of path (no wildcards)
  - suffix: remainder (might have wildcards)

    `/usr/l*/a*`
- Prefix will eventually be the command argument
  - Initially invoked with an empty prefix

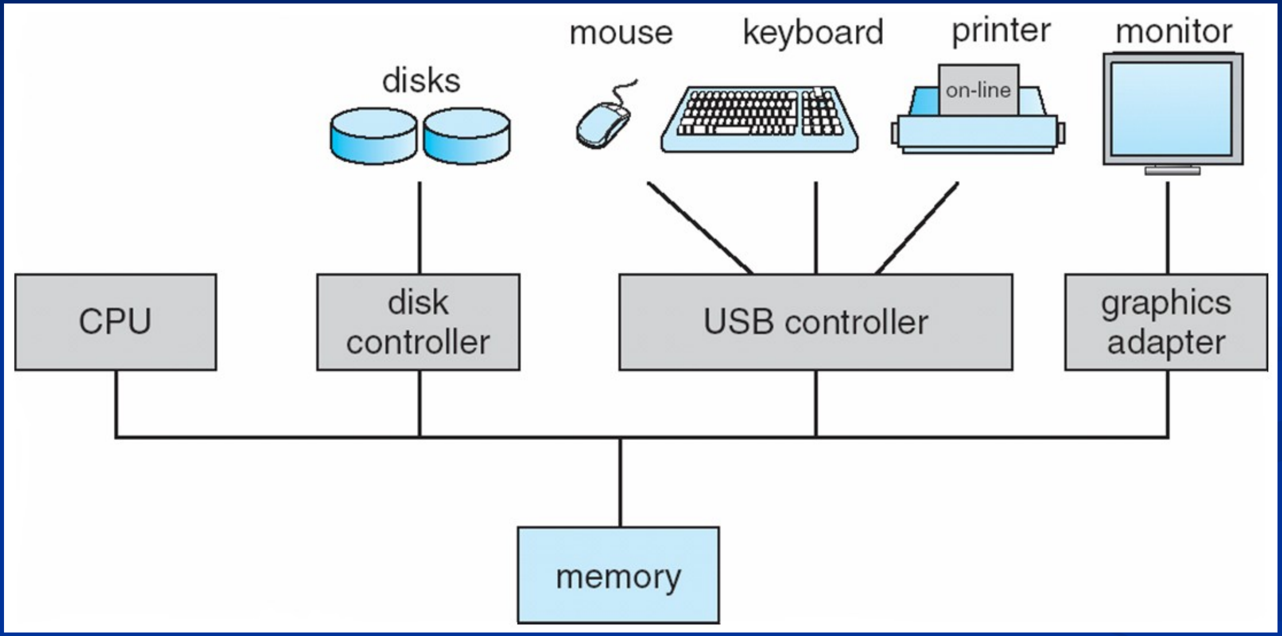
```
#define MAXFILENAME 1024
void expand_wildcards(char * prefix, char *suffix) {
  if (suffix[0] == 0) {
  // suffix is empty. Put prefix in argument.
    insert_argument(strdup(prefix));
    return;
  }
  // Obtain the next component in the suffix
  // Also advance suffix.
  char *s = strchr(suffix, '/');
  char component[MAXFILENAME];
  if (s != NULL) { // Copy up to the first "/"
    strncpy(component, suffix, s-suffix);
    suffix = s + 1;
  }
  else { // Last part of path. Copy whole thing.
    strcpy(component, suffix);
    suffix = suffix + strlen(suffix);
  }
  // Now we need to expand the component
  char new_prefix[MAXFILENAME];
  if ( component does not have '*' or '?') {
    // component does not have wildcards
    sprintf(new_prefix,"%s/%s", prefix, component);
    expand_wildcards(new_prefix, suffix);
    return;
  }
```

```
  // Component has wildcards
  // Convert component to regular expression
  char *exp_buf = regcomp(…)
  char *dir;
  // If prefix is empty then list current directory
  if (prefix is empty) dir =“.”; else dir=prefix;
  DIR *dir = opendir(dir);
  if (d == NULL) return;
  // Now we need to check what entries match
  while ((ent = readdir(d))!= NULL) {
    // Check if name matches
    if (regexec(...ent->d_name) ) {
      // Entry matches. Add name of entry
      // that matches to the prefix and
      // call expand_wildcards(..) recursively
      sprintf(new_prefix,”%s/%s”, prefix, ent->d_name);
      expand_wildcards(new_prefix, suffix);
    }
  }
  close(d);
}// expandWildcard
```

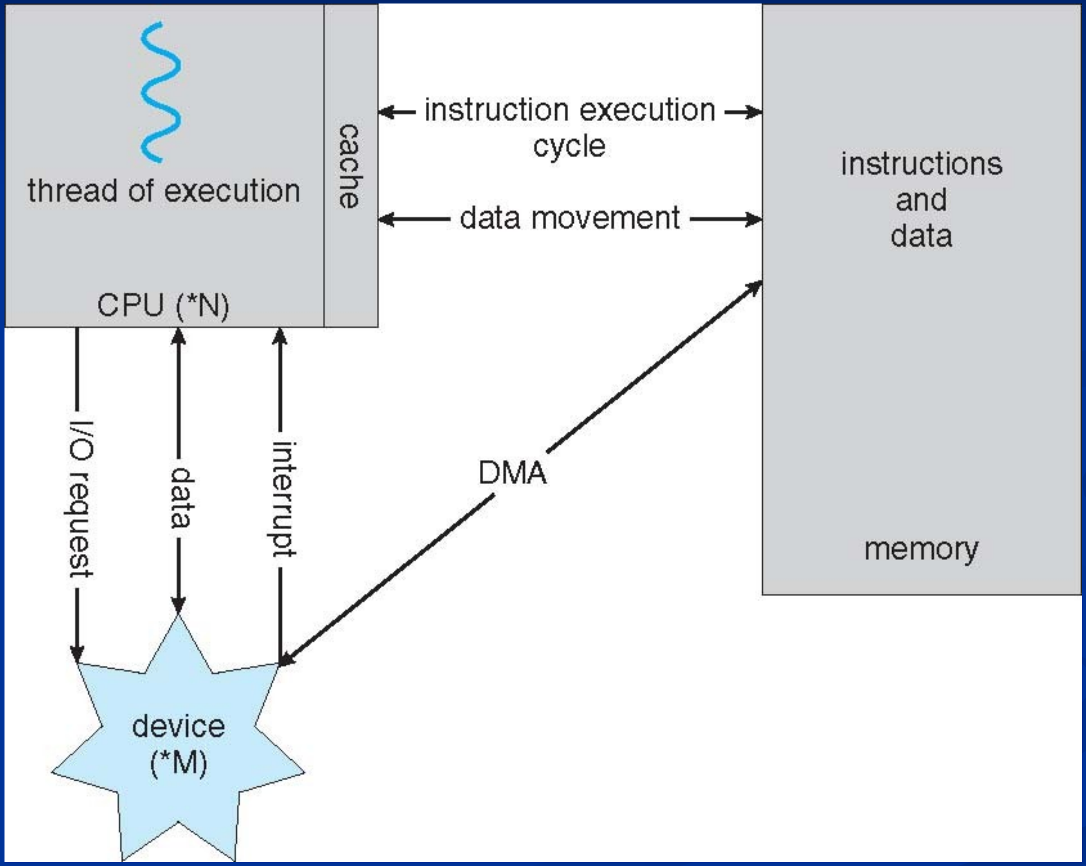
Purdue Trivia

- Purdue has an extensive network of “steam tunnels”
- Connect to almost all buildings on campus
  - Power conduits
  - Fiber/networking
  - Steam
  - Chilled water
  - …and more!
- There are motion sensors!

Computer system organization



How a modern computer works



Harvard architecture

- Idea by Howard Aiken, Harvard physicist, to IBM Nov. 1937
- Built by IBM in Endicott, NY and delivered to Harvard in Feb. 1944 as the Mark 1 computer
- Has separate memories for program (instructions) and data
- Input/output (I/O) to connect to the world
- Processor to carry out the computations

Harvard

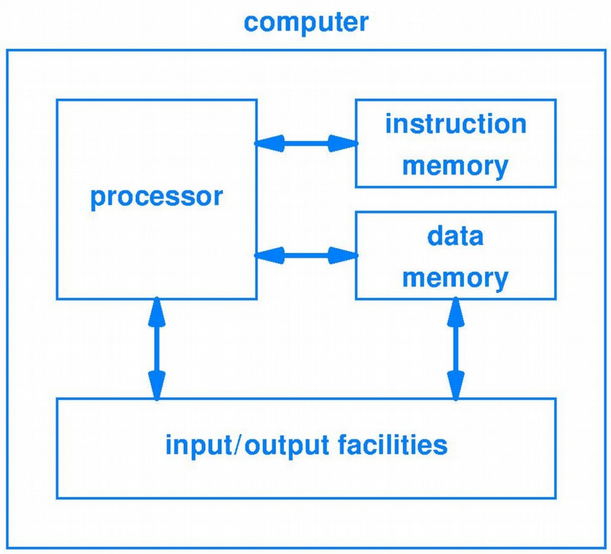


Figure 4.1 Illustration of the Harvard architecture that uses two memories, one to hold programs and another to store data.

(John) Von Neumann architecture

- Developed during his June 1945 train ride from Philadelphia to Los Alamos, NM
- He had programmed the Mark 1 in August 1944
- One memory for both data and program
- Same I/O
- Same processor

Von Neumann

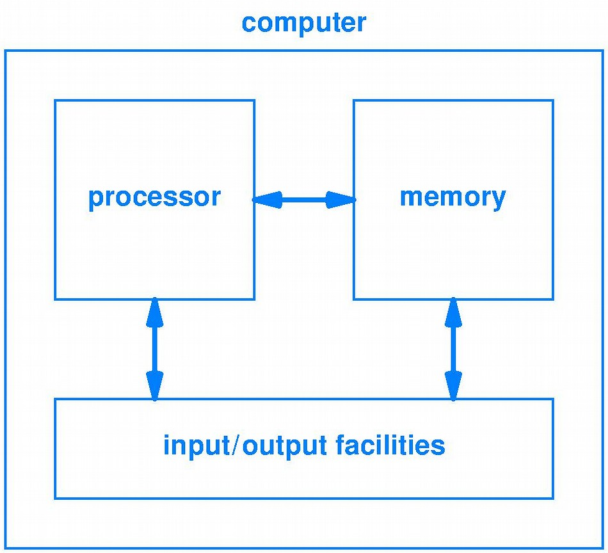
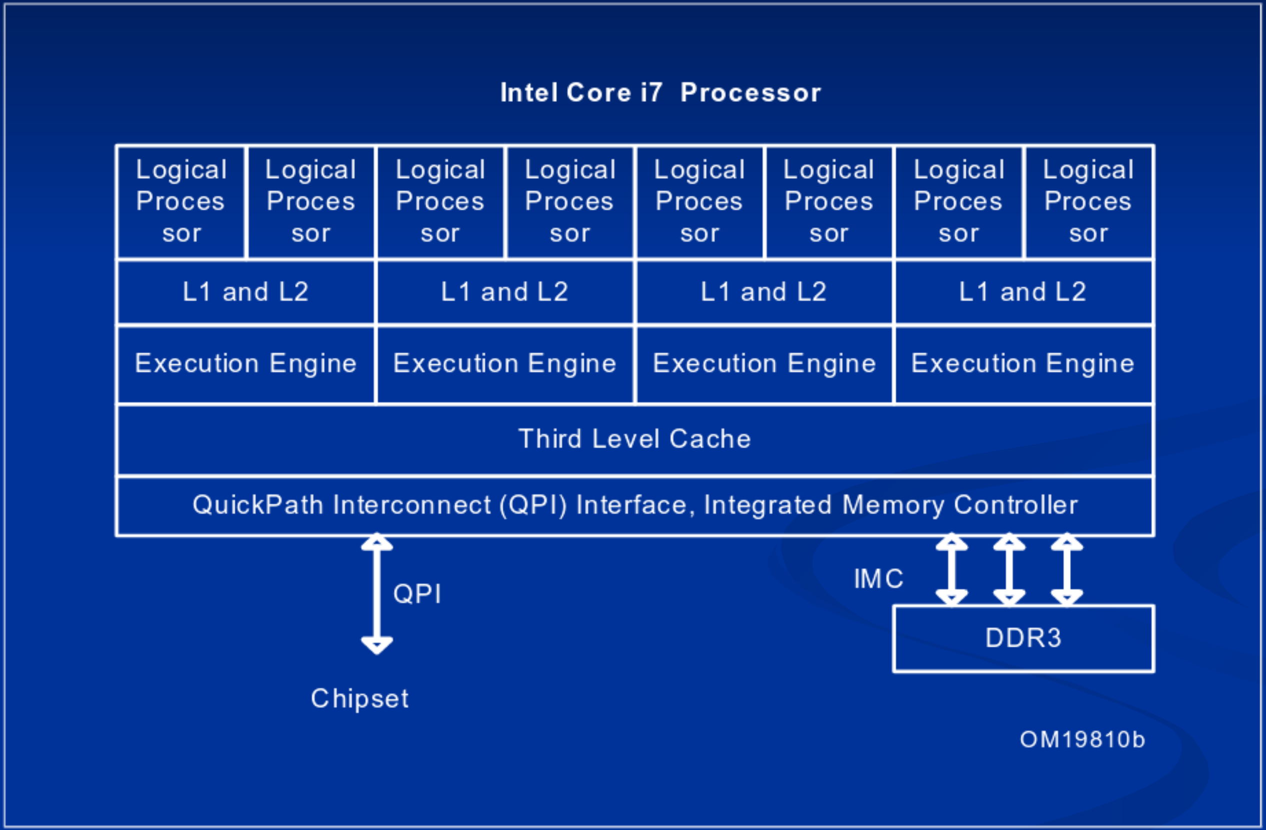


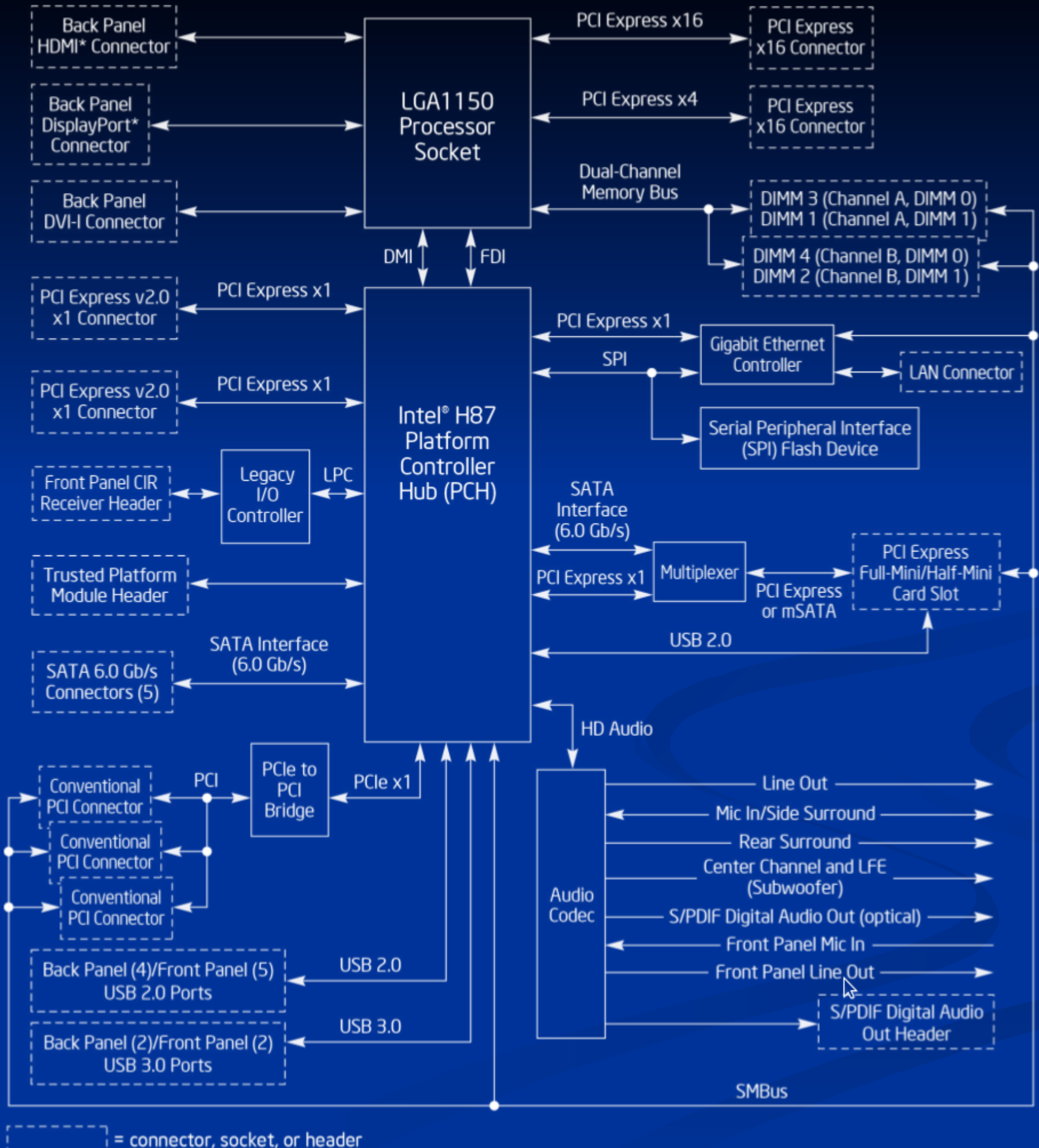
Figure 4.2 Illustration of the Von Neumann architecture. Both programs and data can be stored in the same memory.

von Neumann vs Harvard architectures

- von Neumann
  - Same memory holds instructions and data
  - Single bus between CPU and memory
  - Flexible, more cost effective
- Harvard
  - Separate memories for data and instructions
  - Two busses
  - Allows two simultaneous memory fetches
  - Less flexible, memory is physically partitioned
- Both are stored program computer designs

Intel Core i7 Processor




Back Panel HDMI* Connector
Back Panel DisplayPort* Connector
Back Panel DVI-I Connector
LGA1150 Processor Socket
PCI Express x16
PCI Express x16 Connector
PCI Express x4
PCI Express x16 Connector
Dual-Channel Memory Bus
DIMM 3 (Channel A, DIMM 0)
DIMM 1 (Channel A, DIMM 1)
DIMM 4 (Channel B, DIMM 0)
DIMM 2 (Channel B, DIMM 1)
DMI
FDI
PCI Express v2.0 x1 Connector
PCI Express x1
PCI Express v2.0 x1 Connector
PCI Express x1
Intel® H87 Platform Controller Hub (PCH)
PCI Express x1
Gigabit Ethernet Controller
SPI
LAN Connector
Serial Peripheral Interface (SPI) Flash Device
Front Panel CIR Receiver Header
Legacy I/O Controller
LPC
SATA Interface (6.0 Gb/s)
Multiplexer
PCI Express x1
PCI Express or mSATA
PCI Express Full-Mini/Half-Mini Card Slot
Trusted Platform Module Header
USB 2.0
SATA Interface (6.0 Gb/s)
SATA 6.0 Gb/s Connectors (5)
HD Audio
Conventional PCI Connector
PCI
PCIe to PCI Bridge
PCIe x1
Conventional PCI Connector
Conventional PCI Connector
Audio Codec
Line Out
Mic In/Side Surround
Rear Surround
Center Channel and LFE (Subwoofer)
S/PDIF Digital Audio Out (optical)
Front Panel Mic In
Front Panel Line Out
Back Panel (4)/Front Panel (5) USB 2.0 Ports
USB 2.0
S/PDIF Digital Audio Out Header
Back Panel (2)/Front Panel (2) USB 3.0 Ports
USB 3.0
SMBus
= connector, socket, or header

Direct memory access

- DMA allows other hardware subsystems to access main memory without going through the CPU
- Modern systems usually have DMA controller (MMU)
  - Memory address register, byte count, control, etc
  - Responsible for ensuring accesses are properly restrained
    - Attack vector

MMU

- Responsible for “refreshing” DRAM
- Translates virtual memory addresses to physical addresses
- Sometimes part of CPU
- Sometimes not
  - Northbridge for Intel until recently
  - I7/i5 have an Integrated Memory Controller (IMC)

Questions?