Embedded Program Design

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Components for Embedded Programs
State Machines

- State machine keeps internal state as a variable, changes state based on inputs.

- Uses:
  - Control-dominated code
  - Reactive systems (e.g., user interfaces)
State Machine Example

- Inputs/outputs (- : no action)
C Implementation

```c
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3

switch (state) {
    case IDLE:  if (seat) state = SEATED;
                break;
    case SEATED: if (belt) state = BELTED;
                 else if (timer) state = BUZZER;
                 break;
    case BELTED: if (!seat) state = IDLE;
                 else if (!belt) state = SEATED;
                 break;
    case BUZZER: if (belt) state = BELTED;
                  else if (!seat) state = IDLE;
                  break;
}
```
Circular Buffers (1)

- Commonly used in signal processing
  - New data constantly arrives
  - Each data has a limited lifetime
- Use a circular buffer to hold the data stream
Circular Buffers (2)

Data stream

Circular buffer

$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$

$t_1$ $t_2$ $t_3$

$x_5$ $x_6$ $x_3$ $x_4$
Circular Buffers (3)

- Indexes locate currently used data, current input data:

  \begin{align*}
  \text{use} & \rightarrow \begin{array}{c}
  d_1 \\
  d_2 \\
  d_3 \\
  d_4 \\
  \end{array} & \text{input} & \rightarrow \begin{array}{c}
  d_4 \\
  d_3 \\
  d_2 \\
  d_5 \\
  \end{array} \\
  \text{time } t & & \text{use} & \rightarrow \begin{array}{c}
  d_2 \\
  d_3 \\
  d_4 \\
  \end{array} & \text{input} & \rightarrow \begin{array}{c}
  d_5 \\
  \end{array} & \text{time } t+1
  \end{align*}
Queues

▪ Elastic buffer: holds data that arrives irregularly
▪ Used in signal processing and event processing
▪ Implementation with a linked list
  • Queue can grow to an arbitrary size
  • Requires dynamic memory allocation
Buffer-based Queues

Empty

Full

Empty

Initial state

enQ (d1);

enQ (d2);
enQ (d3);
...
enQ (d7)

deQ ();

deQ ();
deQ ();
...
deQ ();
deQ ();
#define Q_SIZE 32
#define Q_MAX (Q_SIZE - 1)

int q[Q_SIZE];
int head, tail;

void initqueue () {
    head = 0;
    tail = 0;
}

int wrap (int i) {
    return ((i+1) % Q_SIZE);
}

void enqueue (int val) {
    if (wrap (tail) == head)
        error (QUEUE_FULL);
    q[tail] = val;
    tail = wrap (tail);
}

int dequeue () {
    int retval;
    if (head == tail)
        error (QUEUE_EMPTY);
    retval = q[head];
    head = wrap (head);
    return retval;
}
Control/Data Flow Graph
Models of Programs

- Source code is not a good representation for programs:
  - Clumsy
  - Leaves much information implicit

- Compilers derive intermediate representations to manipulate and optimize the program
Data Flow Graph (DFG)

- Does not represent control
- Models basic block: code with no entry or exit
- Describes the minimal ordering requirements on operations
- Represented in single-assignment form
Single Assignment Form

original basic block

\[
\begin{align*}
x &= a + b; \\
y &= c - d; \\
z &= x \times y; \\
y &= b + d; \\
\end{align*}
\]
	single assignment form

\[
\begin{align*}
x &= a + b; \\
y &= c - d; \\
z &= x \times y; \\
y_1 &= b + d; \\
\end{align*}
\]
DFGs and Partial Orders

- The DFG defines a partial ordering of the operations in the basic block.

\[
x = a + b; \\
y = c - d; \\
z = x * y; \\
y_1 = b + d;
\]
Control/Data Flow Graph

- CDFG: represents control and data
- Users data flow graphs as components
- Two types of nodes:

Data flow nodes

\[
\begin{align*}
    x &= a + b; \\
    y &= c - d ;
\end{align*}
\]

Decision nodes

\[
\begin{align*}
    \text{cond} & \quad T \quad v1 \quad v4 \\
    \text{F} & \quad v2 \quad v3
\end{align*}
\]
if (cond1) bb1();
else bb2();
bb3();
switch (test1) {
    case c1: bb4(); break;
    case c2: bb5(); break;
    case c3: bb6(); break;
}
For Loop

```c
for (i = 0; i < N; i++)
    loop_body();
```

```
i = 0;
while (i < N) {
    loop_body ();
    i++;
}
```